CHALMERS | GÖTEBORGS UNIVERSITET

Algorithms for Pure Categorical Optimization

Examensarbete för kandidatexamen i matematik vid Göteborgs universitet
Kandidatarbete inom civilingenjörsutbildningen vid Chalmers tekniska högskola

Oskar Eklund
David Ericsson
Astrid Liljenberg
Adam Östberg

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Algorithms for Pure Categorical Optimization

Examensarbete för kandidatexamen i tillämpad matematik
inom matematikprogrammet vid Göteborgs universitet

Oskar Eklund   David Ericsson   Adam Östberg

Kandidatarbete i matematik inom civilingenjörsprogrammet Teknisk matematik
vid Chalmers tekniska högskola

Astrid Liljenberg

Supervisor: Zuzana Nedělková
Examiners: Marina Axelson-Fisk and Maria Roginskaya

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Preface

Each group member kept an individual time log during the project. We also kept a group log with notes from our group meetings and the meetings with our supervisor.

Work process

During the project we have had group meetings approximately twice per week, as well as meetings with our supervisor every 1-2 weeks. During the group meetings we decided on tasks to be done until the next meeting, and followed up and gave feedback on work done since the last meeting.

All group members have been involved in and kept up to date on all parts of the project during the process, but we did also assign main areas of responsibility for each group member. Oskar was responsible for the report, David was responsible for the theory and literature studies, Astrid was assigned the role of project coordinator, and Adam was responsible for the MATLAB code.

During the work process, David has lead the literature studies. Oskar did much of the work on and wrote the code for the global search algorithm. Astrid did everything related to the genetic algorithm, including coding the algorithm. The numerical results were generated by Adam, who also wrote the code for the local search algorithm as well as the code used for benchmarking.

All group members contributed with literature studies and creative input when developing the neighborhoods and algorithms. The discussion and conclusion parts of the report are the results of group discussions. All important decisions were discussed and made during group meetings.

Report

The writing of different sections of the report was divided between the group members. The authors of each section are presented below, however, everyone has contributed with ideas, comments, and proofreading of each section.

- **Oskar Eklund:** 3.1 Previous research, 3.2 Local Search, 3.3 Global Search, 4.1 Test problems, 5.2 Global Search Algorithm
- **David Ericsson:** 2 Mathematical optimization, 4.2 Benchmarking, 6 Results
- **Astrid Liljenberg:** Abstract, 1.4 Outline, 3.4 Genetic algorithm, 5.3 Genetic algorithm
- **Adam Östberg:** 4 Assessment methodology, 5.1 Local search algorithm, 5.2 Global search algorithm, 5.4 Benchmarking, 6 Results

The remaining sections, not mentioned above, were co-authored during group sessions.

Acknowledgements

We would like to thank our supervisor Zuzana Nedělková for the great guidance and insightful advice she has provided us during the course of the project.

Populärvetenskaplig sammanfattning

Optimering handlar om att försöka finna bästa möjliga lösning på ett problem. Inom matematisk optimering innebär detta att man har en *funktion*, som beroende på de val man gör i problemet, ger ett värde. Det är detta värde man antingen vill minimera eller maximera. Man kan även sätta upp villkor för hur en lösning får lov att se ut. Ett känt exempel på ett optimeringsproblem är *kappsäcksproblemet*. En person har då en kappsäck och ett urval av föremål att fylla kappsäcken med. Varje föremål har ett värde, men kappsäcken har en begränsat utrymme som gör det omöjligt att packa ner samtliga föremål. Problemet är alltså att bestämma vilka föremål personen ska packa ner i kappsäcken för att den ska få så stort värde som möjligt. I detta exempel utgörs funktionen av det sammanlagda värdet av föremålen i kappsäcken, vilket ska maximeras, och villkoren utgörs av kappsäckens utformning och volym.

Att lösa optimeringsproblem är av stort intresse för samhället, då vi har en begränsad mängd resurser som vi vill använda på ett effektivt sätt. Av denna anledning har det under lång tid bedrivits forskning inom ämnet optimering. I dagens datoriserade samhälle handlar denna forskning ofta om att utveckla *algoritmer*, som med hjälp av datorer kan lösa olika problem. I de flesta matematiska optimeringsproblem utgörs valen som funktionen beror på, de så kallade *beslutsvariablerna*, av tal. Dessa tal har en naturlig ordning vilket innebär att det för varje par av tal är möjligt att jämföra dem och säga att det ena är *större än*, *lika med*, eller *mindre än* det andra. För denna typ av problem finns det många kända algoritmer som används för att hitta lösningar. Men det är inte alltid fallet att variablerna är tal, exempelvis om målet är att maximera värdet av ett hus genom att måla om det i en populär färg. Då handlar det istället om färger, vilka saknar en naturlig ordning eftersom det inte finns ett självklart sätt att avgöra om till exempel röd är större än grön. Detta är vad *kategorisk optimering* handlar om; att lösa optimeringsproblem med variabler som inte har någon naturlig ordning.

Volvo GTT bedrev åren 2012-2017 ett forskningsprojekt vid namn TyreOpt som behandlade optimeringsproblemet att välja däck till lastbilar i syfte att minimera bränsleförbrukningen. Detta optimeringsproblem är kategoriskt eftersom däck, likt färger, inte har någon naturlig ordning. Att ett stort företag som Volvo investerar i ett sådant projekt är en god indikation på att kategoriska optimeringsproblem är värda att studera. Vi har därför ägnat vårt arbete till kategorisk optimering, där vi har studerat befintliga algoritmer och även med hjälp av dessa utvecklat egna algoritmer för den här typen av problem.

Vårt arbete har kretsat kring tre algoritmer: en *lokalsökningsalgoritm*, en *globalsökningsalgoritm*, och en *genetisk algoritm*. Lokalsökningsalgoritmen letar inte nödvändigtvis efter den optimala lösningen, utan nöjer sig med att hitta en lösning som är bättre än alla andra lösningar i dess lokala *omgivning*. Omgivningar kan definieras på olika sätt, i problemet med husfärgerna skulle en omgivning till färgen röd exempelvis kunna inkludera färgerna rosa och orange, eftersom de i någon mening ligger nära färgen röd. Globalsökningsalgoritmen, som vi har utvecklat baserat på lokalsökningsalgoritmen, söker å andra sidan efter den verkligt optimala lösningen av problemet. Både lokal- och globalsökningsalgoritmen kräver en omgivningsdefintion för att användas, något som vi också har studerat i detta arbetet. Slutligen så är den genetiska algoritmen en algoritm inspirerad av den biologiska evolutionen, med mekanismer som till exempel naturligt urval och genmutation. Vi har implementerat alla tre algoritmerna i programmeringsspråket MATLAB.

Vidare har vi valt ut lämpliga problem att testa algoritmerna på, för att på så sätt kunna analysera hur bra de presterar. Vi har använt oss av ett påhittat, rent matematiskt problem, och ett fysikaliskt problem som involverar en balk. Genom att testa hur bra algoritmerna löser problemen har vi kunnat dra slutsatser om hur de kan användas i fortsättningen. Exempelvis har vi sett att lokalsökningsalgoritmen med avseende på en viss omgivningsdefinition, som vi kallar *kategorisk omgivning*, fungerade väldigt bra i jämförelse med den genetiska algoritmen. Vi har även sett att med en annan omgivningsdefinition, som vi kallar *diskret omgivning*, fungerar globalsökningsalgoritmen bättre än lokalsökningsalgoritmen. En nackdel vi har kunnat se med globalsökningsalgoritmen är att den i vissa fall tar förhållandevis lång tid för datorn att utföra.

Abstract

Optimization problems with categorical variables are common for example in the automotive industry and other industries where mechanical components are to be selected and combined in favorable ways. The lack of a natural ordering of the decision variables makes categorical optimization problems generally more difficult to solve than the discrete or continuous problems. Thus it is important to develop methods for solving categorical optimization problems. This report presents three different algorithms that can be used for solving categorical optimization problems: a local search algorithm, a global search algorithm, and a genetic algorithm. In addition, two different neighborhood definitions to use with the local search algorithm are presented, a categorical one, and a discrete one. The algorithms were implemented in MATLAB and were tested on two different categorical optimization problems: an artificial problem, and a beam problem. The algorithms developed were applied to a large number of instances of the problems and their performance was evaluated using performance profiles and data profiles. The local search algorithm equipped with the categorical neighborhood outperformed the other algorithms considered.

Sammanfattning

Optimeringsproblem med kategoriska variabler är vanligt förekommande exempelvis inom bilindustrin och andra industrier där mekaniska komponenter ska väljas ut och kombineras på gynnsamma sätt. Avsaknaden av naturlig ordning på beslutsvariablerna gör att kategoriska optimeringsproblem oftast är svårare att lösa än diskreta eller kontinuerliga problem. Det är därför viktigt att ta fram metoder som löser kategoriska optimeringsproblem. Den här rapporten presenterar tre olika algoritmer som kan användas för att lösa kategoriska optimeringsproblem: en lokalsökningsalgoritm, en globalsökningsalgoritm, och en genetisk algoritm. Dessutom presenteras två olika omgivningsdefintioner att använda ihop med lokalsökningsalgoritmen, en diskret, och en kategorisk. Algoritmerna implementerades i MATLAB och testades på två olika kategoriska optimeringsproblem: ett artificiellt problem, och ett balkproblem. De framtagna algoritmerna applicerades på ett stort antal instanser av testproblemen och deras prestanda utvärderades med hjälp av prestandaprofiler och dataprofiler. Lokalsökningsalgoritmen utrustad med den kategoriska omgivningen presterade bäst av de testade algoritmerna.

Contents

1 Introduction

The field of optimization is concerned with finding the best solution to a given problem. The problems are modelled using mathematical tools and can be divided into different branches based on the domain of their decision variables, for example, continuous, and discrete optimization problems. Some discrete optimization problems have variables that lack natural order, that is, there is no meaningful way to order them that corresponds to a physical meaning. Such discrete variables are referred to as categorical variables. Optimization problems having only categorical variables arise in various real life applications and are called pure categorical problems, e.g., an optimization problem to select components in a mechanical construction, see [1].

The existing optimization methods developed for solving problems with continuous or discrete variables cannot be used to solve the categorical optimization problems. This is due to the fact that the usual definitions, for instance neighborhood, optimality, and continuity are not applicable in the context of categorical variables with no natural ordering. In order to analyze and solve pure categorical problems, alternative definitions and solution methods have to be introduced.

1.1 Background

This project was motivated by the research project TyreOpt ([2]) conducted by Chalmers University of Technology and University of Gothenburg in cooperation with Volvo Group Trucks Technology (GTT). The main aim of the research project was to reduce the fuel consumption of heavy duty trucks by optimizing the tire selection. The choice of tires for each axle of the truck, that is, discrete variables with no natural ordering, were the categorical decision variables of the tires selection problem.

Optimization methods and vehicle dynamics models to select the tires when described by their inflation pressure, diameter, and width were developed in [2]. A possible way to improve the methodology developed is to somehow also consider the tire patterns when selecting the tires. Photos of tires are supplied by Volvo GTT, which can be found in Appendix A, with the purpose to find suitable parameterizations of the tire tread patterns allowing for the tire patterns to be considered as additional variables of the tire selection problem.

1.2 Purpose and aim

The purpose of this bachelor project is to analyze categorical optimization problems and to study and develop algorithms to solve such problems. In particular, the aim is to assess how distance and neighborhoods can be defined for categorical variables, and given suitable definitions of these notions, how a categorical optimization problem can be solved efficiently. In order to do this, several definitions of distance and neighborhood for categorical variables are explored with the intention to develop algorithms solving specific categorical optimization problems. The algorithms are developed utilizing existing approaches from the literature studied and our own ideas. In addition, they are implemented in MATLAB and their performance are assessed on different test problems.

1.3 Scope

The project is restricted to the study of pure categorical problems, not considering mixed variable programs (MVP) where the variables may be of both the continuous and discrete (categorical) type. In the theoretical part of the project where we discuss distances and neighborhoods, we do not intend to come up with a completely new definition, but rather review current ones and possibly modify them. We further restrict our project to analyzing three algorithms, called *local search*, *global search* and *genetic algorithm*. For the implementation we restrict the work to two different types of test problems, called the *beam problem* and the *artificial problem*, for testing the performance of the three algorithms. In addition, no ethical aspects connected to project that would require further analysis have been identified and will therefore not be discussed in the report.

1.4 Outline

In Section 2 we begin with an introduction to mathematical optimization, defining the concepts of both the local and the global optimum. The optimization problems with categorical variables are subsequently introduced and it is explained how they differ from the optimization problems with discrete or continuous variables. Then, distances and neighborhoods for categorical variables are discussed.

Section 3 introduces the three algorithms developed for solving optimization problems with categorical variables, including a local search algorithm for finding local optima, a global search algorithm with the intention to find global optima, as well as a genetic algorithm.

In Section 4 two categorical optimization problems are presented, an artificial, and a real world optimization problem. It is also described how the so-called data and performance profiles can be used to assess the performance of the algorithms. In Section 5 we describe how the algorithms were implemented in MATLAB.

The results of numerical tests of the algorithms when applied to the test problems are presented in Section 6, and discussed in Section 7, as well as the implications for the TyreOpt research project. In Section 8 we draw conclusions regarding the algorithms developed for solving optimization problems with categorical variables. Also, we draw conclusions about the TyreOpt project.

2 Mathematical optimization

Mathematical optimization is the discipline concerned with finding the best solution to a given problem. To achieve this, a mathematical model of the problem is set up, and using mathematical methods an optimal solution is searched for. To facilitate the understanding of the technicalities of the report, this section begins by introducing relevant concepts in optimization on a general level. Then, a more detailed account of categorical optimization is given.

2.1 Overview

We begin by setting up the mathematical environment for general optimization problems. In order to describe the choices that can be made in the problem, a space of decision variables Y is introduced. Each element $\mathbf{y} \in Y$ represents a choice one can make in the problem. To be able to determine which choice is the best, an objective function $f : Y \to \mathbb{R}$ is needed, which measures how well a decision $\mathbf{y} \in Y$ solves the problem. In addition, there may be some constraints that need to be satisfied, specifying a subset $\mathcal{C}$ of the decision space Y. The set $\mathcal{C}$ is often described in terms of constraint functions $g_i : Y \to \mathbb{R}$ satisfying $g_i(\mathbf{y}) = 0$ or $g_i(\mathbf{y}) \leq 0$, where i belongs to some index set $\mathcal{I}$. Equipped with these notions, a general optimization problem may be formulated mathematically as

$$\begin{aligned} \underset{\mathbf{y}\in Y}{\text{minimize}} \quad & f(\mathbf{y}) \\ \text{subject to} \quad & \mathbf{y} \in \mathcal{C}, \end{aligned} \tag{1}$$

where the minimization of the objective function f is taken by convention, since any maximization problem may be converted into a minimization problem by changing the sign of f. We will henceforth use the word optimum and minimum interchangeably. The goal is to find a feasible point, that is, a decision $\mathbf{y} \in Y$ that fulfills the constraints, i.e., $\mathbf{y} \in \mathcal{C}$, such that there is no other feasible point with a lower objective value. Such a point is called a global optimum.

Definition 2.1. (Global optimality) A point $\mathbf{y} \in Y \cap \mathcal{C}$ is said to be a global optimum for the problem (1) if

$$f(\mathbf{y}) \leq f(\mathbf{x}), \quad \forall \mathbf{x} \in Y \cap \mathcal{C}.$$

In some problems a global minimum may not exist due to unboundedness of the objective function over the decision space. Even if a global minimum exists, it may be difficult to find it. However, a point may still have the lowest objective value in a small region around itself. This is the notion of local optimality.

Definition 2.2. (Local optimality) A point $\mathbf{y} \in Y$ is said to be a local minimum for the problem (1) if

$$f(\mathbf{y}) \leq f(\mathbf{x}), \quad \forall \mathbf{x} \in \mathcal{N}(\mathbf{y}) \cap \mathcal{C},$$

where $\mathcal{N}(\mathbf{y})$ is a neighborhood of $\mathbf{y}$.

A neighborhood is a set map $\mathcal{N} : Y \to \mathcal{P}(Y)$ which assigns to each point $\mathbf{y}$ in the decision space a subset of points $\mathcal{N}(\mathbf{y}) \subset Y$, i.e., $\mathcal{N}(\mathbf{y})$ belongs to the power set[1] of Y, defining the points in this subset to be similar or close to $\mathbf{y}$. The neighborhood $\mathcal{N}(\mathbf{y})$ may be chosen differently depending on the problem, in order to account for what points should be close to each other. For example, when $Y \subseteq \mathbb{R}^n$ the neighborhood is usually taken as a Euclidean ball, that is, $\mathcal{N}(\mathbf{y}) = \{\mathbf{x} \in Y : \|\mathbf{x} - \mathbf{y}\| < \varepsilon\}$ for some $\varepsilon > 0$. A more thorough discussion of neighborhoods can be found in [3].

2.2 Optimization with discrete and categorical variables

In the previous section general optimization problems were considered, without specifying the underlying decision space nor assumptions on the objective function and the constraints. A *discrete* optimization problem is a problem in the form (1) with $Y \subseteq \mathbb{Z}^n$. Solving discrete optimization problems can easily become cumbersome. Discrete decision spaces can quickly become enormous in size. Consider for example a decision problem with 100 binary variables. For such a problem there are 2^{100} different feasible solutions. The most naive way of solving optimization problems is by means of enumeration, that is, computing the objective value for all feasible points in order to check which has the lowest. Such an approach is not appropriate when the decision space becomes enormous. For continuous decision spaces, such as $Y \subseteq \mathbb{R}^n$, that are of infinite size, tools from calculus have been used to develop algorithms that take shortcuts in order to avoid the enumeration; we refer the reader to [4] for a more in depth description of such algorithms. For discrete cases, development of similar techniques has not been as successful, and the discrete optimization algorithms require much higher effort ([5]).

Categorical optimization problems are ones where the variables represent configurations in some manner. Examples are fixed configuration problems ([6]) and catalog-based design problems ([7]). A fixed configuration problem is when a product with certain components have been decided for, but it remains to choose the dimensions of each component. The catalog-based design problem can be seen as more general, where one chooses components in order to design a tailor-made product. The defining property for such problems is the fact that the decision variables have no natural order. In other words, there is no natural way to compare two elements x and y, as is the case for integers and real numbers.

A categorical optimization problem can be transformed into a discrete one by means of *numerization* as described in [8]. Suppose we have a problem where we need to choose n components, where for the j-th component there are m_j possible choices, $j = 1, \ldots, n$. We may then label the choices for the j-th component by $\{1, \ldots, m_j\} =: X^j$. Hence, the whole set of configurations can be written as $X = X^1 \times \cdots \times X^n$, and each configuration can be denoted $\mathbf{x} = (x_1, \ldots, x_n)$ with $x_j \in X^j$. Thus, a categorical optimization problem may be stated as

$$\underset{\mathbf{x} \in X}{\text{minimize}} \quad f(\mathbf{x}). \tag{2}$$

After a numerization, X can be considered a subset of $\mathbb{Z}^n$. Since the list of configurations can be sorted in an arbitrary way, the actual ordering of the choices in the numerization does not necessarily have any real physical meaning. Given all possible orderings of the configurations, one gets a family of discrete problems that all correspond to the same categorical problem.

[1]The power set of a set A, denoted $\mathcal{P}(A)$, is the set of all subsets of A.

We now want to define what it means for two configurations $\mathbf{x}, \mathbf{y} \in X$ to be close to each other. However, since it is not certain that the ordering in the discrete problem has some connection to the actual structure of the problem, it makes sense to consider a metric which is invariant under different numerizations. The Hamming distance is such a metric ([8]).

Definition 2.3. (Hamming distance) The Hamming distance between two configurations $\mathbf{x}, \mathbf{y} \in X$ is defined as $d_H(\mathbf{x}, \mathbf{y}) = \text{card}\{i \in \{1, \dots, n\} : x_i \neq y_i\}$.[2]

In terms of the Hamming distance, points in the decision space are considered to be close to each other if they differ on few components. With this definition, two points will indeed be close to each other no matter which ordering is chosen in the numerization. Given this metric, a categorical neighborhood may be defined as follows.

Definition 2.4. (Categorical k-neighborhood) Consider the categorical decision space X with the Hamming metric. The categorical k-neighborhood of $\mathbf{x} \in X$ is defined as $\mathcal{N}_c^k(\mathbf{x}) = \{\mathbf{y} \in X : d_H(\mathbf{x}, \mathbf{y}) \leq k\}$.

Since the categorical problem is defined on $\mathbb{Z}^n$ after a numerization, one can also consider a discrete neighborhood. This may be appropriate when there is some underlying structure in the ordering of the configurations. An example is the fixed-configuration problem previously mentioned.

Definition 2.5. (Discrete t-neighborhood) The discrete t-neighborhood of $\mathbf{x} \in X$ is defined as $\mathcal{N}_d^t(\mathbf{x}) = \{\mathbf{y} \in X : \sum_{i=1}^n |x_i - y_i| \leq t\}$.

The categorical k-neighborhood contains all configurations such that k components of $\mathbf{x}$ may be changed. In the discrete neighborhood, we do not only consider points on the lines parallel to the axes, but points that can be reached in t unit steps. Examples of the categorical 1-neighborhood and the discrete 2-neighborhood are illustrated in Figure 1 and Figure 2 for a categorical space with two decision variables, each having five configurations.

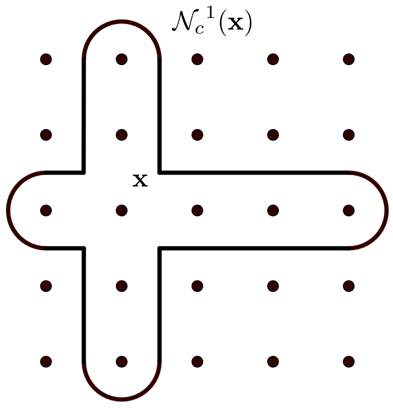


Figure 1: Categorical neighborhood, $k = 1$

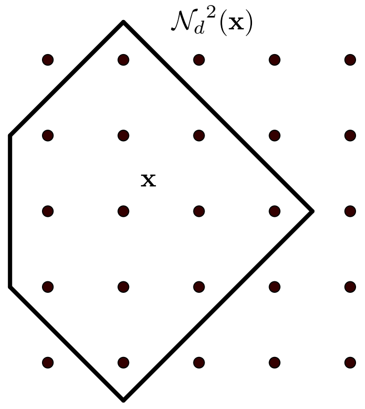


Figure 2: Discrete neighborhood, $t = 2$

Now when the notion of neighborhood for categorical spaces is defined, we can proceed to define optimality conditions. While global optimality for a categorical problem is the same as in the general case, local optimality is defined as follows.

Definition 2.6. (Local optimality with categorical k-neighborhood) A point $\mathbf{x} \in X$ is said to be a *local categorical minimum* for the problem (2) if $f(\mathbf{x}) \leq f(\mathbf{y})$ for all $\mathbf{y} \in \mathcal{N}_c^k(x)$.

Definition 2.7. (Local optimality with discrete t-neighborhood) A point $\mathbf{x} \in X$ is said to be a *local discrete minimum* for the problem (2) if $f(\mathbf{x}) \leq f(\mathbf{y})$ for all $\mathbf{y} \in \mathcal{N}_d^t(\mathbf{x})$.

Local optima are not guaranteed to be global optima. However, many optimization problems of practical relevance exhibit multiple local optima, thus demanding the use of global optimization techniques for their solution. Both local and global optimization techniques for categorical problems are described in Section 3.

[2]The cardinality of a finite set A, denoted card A, is the number of elements in the set.

3 Algorithms for categorical optimization problems

In this section we commence with an account of the previous research related to our project. Then, three algorithms for solving categorical optimization problems, inspired by the previous research, are laid out. The three algorithms are *local search*, *global search*, and *genetic algorithm.*

3.1 Previous research

Optimization problems with categorical variables can be approached in different ways. One way is discrete optimization algorithms, see for example [9] and [10]. In [9] we find a method for solving discrete optimization problems with help of an auxiliary function. In each iteration this method moves from a local minimum of the objective function to another, better one, with help of the auxiliary function. In [10] a method based on splitting of the set of configurations is presented. For a global descent approach, see [8]. Genetic algorithms have been previously used in many component selection and catalog design problems, see for example [11], [7], and [6]. In our work we utilize the local search algorithm described in [8] and come up with a suggestion for a global search.

In [8] a discrete global descent approach of solving categorical optimization problems is used by extending the discrete global descent method described in [9] to categorical problems. There are two suggested extensions: *categorical local search* and *sorting.* Categorical local search is also used in our work, and described in the next section. Sorting is a procedure that is based on the fact that different permutations of the variables of a categorical problem lead to categorically equivalent problems. The idea with finding good sortings is to get well-behaved numerical problems which are easy to solve with methods for numerical optimization.

3.2 Local search

The following algorithm for finding local minima of a categorical optimization problem (2) is presented in [8]. This algorithm requires a neighborhood definition and a starting point $\mathbf{x}^0$ in the set of configurations. We choose the starting point $\mathbf{x}^0$ randomly from the set of feasible configurations.

The idea of this algorithm is to, in each iteration, loop through the neighborhood of the current point and search for a point with lower objective value. Let f be the objective function and let $\mathcal{N}(\mathbf{x})$ be the neighborhood of a configuration $\mathbf{x}$. The steps of the local search algorithm are presented in Algorithm 1. The algorithm terminates if we for some $\mathbf{x}$, have looped through the neighborhood without having found any $\mathbf{y} \in \mathcal{N}(\mathbf{x}) \setminus \{\mathbf{x}\}$ with $f(\mathbf{y}) < f^*$. That is, if we in one iteration loop through the neighborhood and no configuration in the neighborhood of the current configuration has lower objective value than the current configuration. By having this termination criteria we guarantee that we find a local minimum, according to Definition 2.2. In the worst case scenario we have to compute the objective function for each point in the set of configurations.

Algorithm 1 Local Search

```
 1: Choose starting point $\mathbf{x} = \mathbf{x}^0$ and let $f^* = f(\mathbf{x})$.
 2: Compute $\mathcal{N}(\mathbf{x})$.
 3: while Termination criterion is not fulfilled
 4:     for $\mathbf{x}^i \in \mathcal{N}(\mathbf{x}) \setminus \{\mathbf{x}\}$
 5:         if $f(\mathbf{x}^i) < f^*$
 6:             Let $f^* = f(\mathbf{x}^i)$ and return to Step 2 with $\mathbf{x} = \mathbf{x}^i$.
 7:         end
 8:     end
 9: end
10: Return the locally optimal point $\mathbf{x}^* = \mathbf{x}$ with objective value $f^*$.
```

3.3 Global search

In this section we describe a suggestion for a global search algorithm for categorical optimization problems on the form (2). Although the name suggests this algorithm finds a global minimum, we want to stress that this is an extension of the local search and does not guarantee that a global minimum is found. The extension is focused on finding a good starting point $\mathbf{x}^0$, since Algorithm 1 does not provide any computation based suggestions on how to do this. We suggest a procedure on how to select several starting points for Algorithm 1 resulting in a global search algorithm denoted Algorithm 2.

Let us assume that we have the given optimization problem as in (2). As in Algorithm 1, we require a neighborhood defined on X and we also need a set of starting points. Let $S \subset X$ be the set of starting points.

The first 6 steps of Algorithm 2 deals with performing local searches with different starting points, where the starting points are suggested to be chosen randomly in X. By doing this we are later able to construct a new starting point which is not randomly chosen. The vector $\mathbf{u} = (\mathbf{u}^1, \dots, \mathbf{u}^{|S|})$ consists of points where $\mathbf{u}^i \in X$ is the local optimum returned from Algorithm 1 for $i = 1, \dots, |S|$ and $\mathbf{v} = (v_1, \dots, v_{|S|})$ is a vector of objective values where $v_i = f(\mathbf{u}^i)$ for $i = 1, \dots, |S|$.

In Algorithm 2 we aim to get a better starting point $\mathbf{y}$ than the starting points in S in the sense that it is closer to the global optimum of the problem and thus, more often finds lower values. In order to get such a $\mathbf{y}$, we construct a vector of weights $\mathbf{w} = (w_1, \dots, w_{|S|})$ in Step 7 of Algorithm 2. To construct $\mathbf{w}$, we need the vector $\mathbf{v} = (v_1, \dots, v_{|S|})$ of objective values of the points in S. We define the index vector $\mathbf{a} = (a_1, \dots, a_{|S|})$ by letting $a_j = |S|$ for the lowest value v_j of $\mathbf{v}$, $a_k = |S| - 1$ for the second lowest value v_k of $\mathbf{v}$ and so on up to $a_\ell = 1$ for the highest value v_ℓ of $\mathbf{v}$. For example, with $|S| = 4$, if $\mathbf{v} = (0.13\ 0.01\ 1.57\ 2.31)$ then $\mathbf{a} = (3\ 4\ 2\ 1)$. We then compute $\mathbf{w}$ as

$$w_i = \frac{a_i}{\sum_{j=1}^{|S|} a_j}, \quad \forall i \in \{1, \dots, |S|\}. \tag{3}$$

We observe the following properties of $\mathbf{w}$:

$$\sum_{k=1}^{|S|} w_k = 1, \tag{4}$$

$$w_i \geq w_j \iff v_i \leq v_j, \quad \forall i, j \in \{1, \dots, |S|\}. \tag{5}$$

By (5) we can be sure that lower objective values implies higher weights, which is sought since we in our upcoming construction of $\mathbf{y}$ want the points in $\mathbf{u}$ to be weighted such that a point $\mathbf{u}^i$ with lower objective value than another point $\mathbf{u}^j$ has greater impact on $\mathbf{y}$. Equation (4) says that the sum of the weights is equal to 1, and thus it makes sense to refer to them as weights. When we have computed a weight vector $\mathbf{w}$ which satisfies these conditions we are able to construct $\mathbf{y}$ in Step 8 of the algorithm. Let N be the number of elements in a configuration $\mathbf{x} \in X$. Initially we construct $\mathbf{y}$ as

$$y_i := \sum_{k=1}^{|S|} w_k u_k^i, \quad \forall i \in \{1, \dots, N\}. \tag{6}$$

This construction does not guarantee that $\mathbf{y}$ is a feasible configuration. Thus if needed, we finish this construction by simply rounding $\mathbf{y}$ to a feasible configuration to end Step 8. In practice, we do this by letting $y_i = y_i^r$ with

$$y_i^r := \operatorname*{argmin}_{y_k \in X^i} (y_i - y_k), \quad \forall i \in \{1, \dots, N\}. \tag{7}$$

The motivation behind the construction of $\mathbf{y}$ is that we anticipate that it is closer, in the sense of the distance definition given, to the global optimum than just choosing a point randomly out of X. This is due to the fact that each element y_i of $\mathbf{y}$ is computed with respect to the objective values of the outcome of the initial local searches, and in such a way that lower objective value from the initial local search implies stronger impact on y_i, for each y_i. Further, if $\mathbf{y}$ is closer to the global optimum compared to a starting point randomly chosen out of X, we hope that a local search from $\mathbf{y}$ will result in finding a lower value. However, it is not certain that this is the case.

The final step of Algorithm 2 is doing a local search with our constructed $\mathbf{y}$ as starting point. Since we can not be sure if $f(\mathbf{y}^*) < f(\mathbf{u}^i) \;\; \forall i \in \{1, \dots, |S|\}$, we return the point with the lowest objective value of $\mathbf{y}^*$ and the points of $\mathbf{u}$.

Algorithm 2 Global Search

```
 1: Choose a set of starting points $S \subset X$.
 2: Let $\mathbf{v}$ and $\mathbf{w}$ be vectors with $|S|$ elements and let $\mathbf{u}$ be a vector of points, with $|S|$ elements.
 3: for $\mathbf{s}^i \in S$
 4:     Perform Algorithm 1 with starting point $\mathbf{x} = \mathbf{s}^i$.
 5:     Let $\mathbf{u}^i = \mathbf{x}^*$ and $v_i = f(\mathbf{x}^*)$, where $\mathbf{x}^*$ is the point returned from Algorithm 1.
 6: end
 7: Compute $\mathbf{w}$ with respect to $\mathbf{u}$ and $\mathbf{v}$ as in (2).
 8: Compute $\mathbf{y} \in X$ with respect to $\mathbf{u}$ and $\mathbf{w}$ as in (5).
 9: Perform Algorithm 1 with starting point $\mathbf{x} = \mathbf{y}$ and let $\mathbf{y}^* = \mathbf{x}^*$.
10: Return the point $\mathbf{x}^* = \underset{\mathbf{u} \cup \{\mathbf{y}^*\}}{\operatorname{argmin}} f(\mathbf{x})$ with objective value $f(\mathbf{x}^*)$.
```

Note that we can not be sure that Algorithm 2 actually finds a better starting point $\mathbf{y} \in X$ than a randomly chosen starting point in X, since it certainly depends on the structure of the problem. Later sections will show how well Algorithm 2 performs.

3.4 Genetic algorithm

In this section we describe a version of the genetic algorithm presented in [12]. The genetic algorithm does not require a neighborhood definition for the problem at hand. This is a difference compared to previous algorithms, and makes it an interesting alternative approach to solving a categorical optimization problem. It is however important to note that there is no way of knowing how good a solution will be found, not even a local optimum is guaranteed. In that sense the genetic algorithm is more of a heuristic, and is usually only used when classical methods cannot be utilized. Another drawback of the genetic algorithm is that there are a lot of parameter values that can be varied. The best values varies between problems, making it difficult to know in advance what parameter values to choose.

The genetic algorithm encodes a set of points or configurations $\mathbf{x}^j \in X, j = 1, \dots, N$ in the search space as a set of chromosomes $\mathbf{c}^j,\ j = 1, \dots, N$, called a population P of size N. It is important that there is one-to-one correspondence between $\mathbf{x}^j$ and $\mathbf{c}^j$, so that a chromosome can be decoded to find the original configuration or point. Chromosomes are also referred to as individuals. The chromosomes are strings of real or integer values, to which a number of biologically inspired operators are applied to form a new population. Each value of a chromosome is called a gene. There are different variations of the genetic algorithm depending on what operators and variations thereof that are chosen during the implementation. One version of the algorithm will be introduced below, followed by detailed descriptions of the different steps, and it is based largely on the book on stochastic optimization by Wahde ([12]).

Algorithm 3 Genetic algorithm

```
1: Initialize a population $P$ of size $N$ by randomizing chromosomes $\mathbf{c}^j$, $j = 1, \ldots, N$.
2: while Termination criterion is not fulfilled.
3:     for $\mathbf{c}^j \in P$
4:         Using the encoding scheme, decode chromosome $\mathbf{c}^j$ to form configuration $\mathbf{x}^j \in X$.
5:         Evaluate $f(\mathbf{x}^j)$, where $f$ is the objective function.
6:         Based on the objective value, assign a fitness value $\mathcal{F}^j$ to the individual.
7:     end
8:     Let $P_{\text{new}} := \varnothing$ be an empty set of chromosomes.
9:     for $j = 1, \ldots \frac{N}{2}$
10:        Select with replacement two individuals $\mathbf{i}^1$ and $\mathbf{i}^2$ from $P$ using the selection operator.
11:        With probability $p_c$, crossover $\mathbf{i}^1$ and $\mathbf{i}^2$ to form new individuals $\mathbf{i}^1_{\text{new}}$ and $\mathbf{i}^2_{\text{new}}$.
12:        Perform mutation on $\mathbf{i}^1_{\text{new}}$ and $\mathbf{i}^2_{\text{new}}$.
13:        Let $P_{\text{new}} := P_{\text{new}} \cup \{\mathbf{i}^1_{\text{new}}, \mathbf{i}^2_{\text{new}}\}$.
14:    end
15:    Replace an arbitrary individual in $P_{\text{new}}$ with $\mathbf{c}^{\text{elite}} \in P$ such that $\mathcal{F}^{\text{elite}} \geq \mathcal{F}^k$, $k = 1, \ldots, N$.
16:    Let $P := P_{\text{new}}$.
17: end
18: Return the configuration $\mathbf{x}^j \in X$ such that $\mathcal{F}^j \geq \mathcal{F}^k$, $k = 1, \ldots, N$, and objective value $f(\mathbf{x}^j)$.
```

The first thing to decide on is what encoding scheme to use. The purpose of an encoding is to have a translation between a point, or configuration for categorical problems, and a chromosome. An example is *binary encoding*, which was first introduced by Holland ([13]). On a one-dimensional continuous problem, the variable $x \in [-d, d] = X$ is encoded as a chromosome $\mathbf{c} = (c_1, \ldots c_k)$ of optional length k. Each gene c_i is either a 0 or 1, and the corresponding point x is given by

$$x = -d + \frac{2d}{1 - 2^{-k}}(2^{-1}c_1 + \ldots + 2^{-k}c_k).$$

Once an encoding scheme is selected, a population of individuals is randomly generated (Step 1) by uniformly randomizing the values of the genes from the permitted interval or set of values. In the case of binary encoding the set would be $\{0, 1\}$. Based on this initial population, a new generation will be formed.

In order to decide which individuals will be selected to make up next generation's population, we need to evaluate the current population (Step 5) and assign a *fitness value* to each individual (Step 6). To evaluate a chromosome, it must first be decoded using the encoding scheme backwards (Step 4), and then the corresponding configuration is evaluated in the objective function. A higher fitness value should always correspond to a better solution. For maximization problems the fitness value of an individual is usually taken to be the objective value of the configuration, while minimization problem instead usually uses the multiplicative or additive inverse of the objective value.

Based on the assigned fitness values, individuals of the population should now be selected for reproduction (Step 10). Two common methods of selection are *roulette-wheel selection* and *tournament selection*. In the former, the probability of selecting an individual is directly proportional its fitness value, while in the latter, individuals are compared and the one with a larger fitness value is assigned a fixed, higher probability regardless of how much larger the fitness value is.

The selection operator is then repeatedly applied to the population, each time selecting two individuals $\mathbf{i}^1$ and $\mathbf{i}^2$ from P. Note that selection is done with replacement, i.e. the same individual may be selected multiple times. To each pair, the *crossover* operator is applied with some probability p_c (Step 11). The point of the crossover is to combine the two individuals to form two new individuals, hopefully corresponding to better solutions to the optimization problem. Usually one or more crossover points are generated and the genes are swapped accordingly, see Figure 3 for an illustration of two point crossover. In the example, the lengths of the chromosomes are not preserved. If desired, this is fixed by using the same crossover points on both individuals. There is a $1 - p_c$ probability of no crossover occurring, in which case the new pair of individuals is identical to the old.

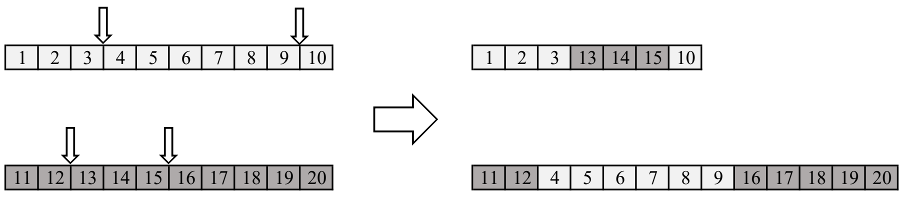


Figure 3: An illustration of non length preserving two point crossover. The small arrows indicate the randomly generated crossover points.

Each gene in each chromosome then has a probability p_{mut} of *mutating* (Step 12). This means that the gene is replaced with a new value. How the new gene is generated varies, the most common method being to uniformly generate it at random.

Finally, the *elitism* operator copies the individual with highest fitness value in the current population and inserts one or a few copies of it into the next generation (Step 15). This makes sure that the best individual is never lost, which could otherwise happen due to bad luck during crossover and mutation.

This process of forming new generations is repeated until some termination criterion is met. It could be that a fixed number of generations have been evaluated, or that some convergence criterion is fulfilled, for example that the optimal value has not improved for a certain number of generations.

4 Assessment methodology

In order to evaluate the algorithms and different neighborhoods described above, the algorithms will be applied to different test problems. In this section, we first describe the test problems consisting of an artificial optimization problem without any physical interpretation and a beam design optimization problem. Then, performance profiles used to assess how well the algorithms perform will be explained.

4.1 Test problems

The first test problem, referred to as the artificial problem, is to

$$\begin{aligned} \underset{\mathbf{x},\mathbf{z}}{\text{minimize}} \quad & \frac{1}{2}\mathbf{z}^T\mathbf{Q}\mathbf{z} + \mathbf{p}^T\mathbf{z} + (\text{diag}(\mathbf{z})\mathbf{z})^T\mathbf{S}\mathbf{z}, \\ \text{subject to} \quad & \mathbf{z}^i = \mathbf{Z}_i(x_i), \quad i = 1, \dots, m, \\ & x_i \in X^i, \qquad\quad i = 1, \dots, m, \end{aligned} \tag{8}$$

where $\mathbf{Q}$ and $\mathbf{S}$ are diagonal matrices with elements $\mathbf{Q}_{ii} \in [-3, 3]$ and $\mathbf{S}_{ii} \in [-3, 3]$ respectively, which are uniformly randomly generated. Further, $\mathbf{p}$ is a row vector with elements $p_i \in [-1, 1]$, which are also uniformly randomly distributed. The set of feasible designs X consists of vectors $\mathbf{x} \in \mathbb{R}^m$ such that the number of feasible designs N_i in each choice domain X^i is randomized from the integer uniform distribution on $[10, 30]$ for each $i = 1, \dots, m$. Further, m is randomized from the integer uniform distribution on $[4, 8]$. Lastly, $\mathbf{Z}$ is a table mapping which is randomly generated such that $\mathbf{Z}_i : X^i \to \mathbb{R}^{N_i}$ maps each $x_i \in X^i$ onto a vector with N_i elements, for each $i = 1, \dots, m$. The mapping $\mathbf{Z}_i$ is represented as a matrix where each element is chosen from the integer uniform distribution on $[5m, 10m]$.

The second test problem stems from an article written by Thanedar and Vanderplaats ([14]), and regards a cantilever beam that is fixed to the wall at its left end and loaded with a constant force F at its right end, see Figure 4. The beam is of length L and consists of K parts, referred to as segments.

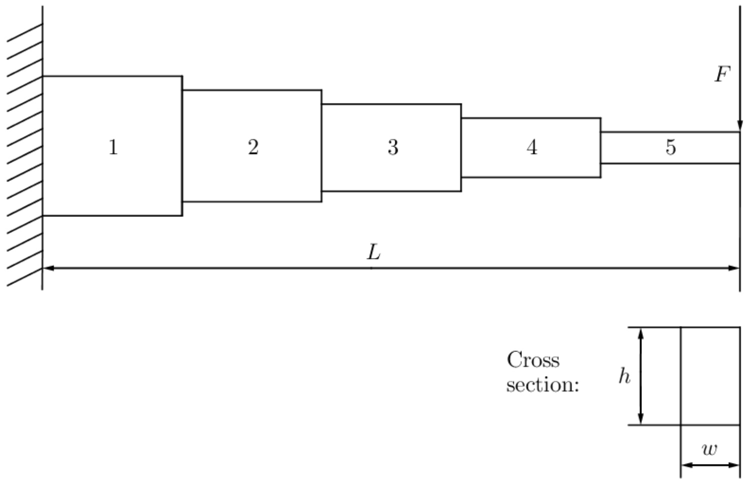


Figure 4: Stepped cantilever beam consisting of five segments loaded with a force at its right end

Each segment is considered to have two design variables, a height h and a width w. The original problem in [14] was to minimize the volume of the beam subject to constraints regarding a maximal allowable deflection and the bending stress in each segment. However, in this project we will consider a simplified version, the problem to minimize the deflection of the beam subject to an aspect ratio between the height and the width of each segment, namely $h \leq 30w$. We consider the set of all configurations for each segment are combinations of heights and widths chosen from the set of M uniformly spaced points in the interval $[450, 600]$ and $[20, 50]$ respectively. Hence, the distance between the points is fixed and will be the length of the intervals divided by the number of configurations, namely $\ell_1 := \frac{150}{M}$ and $\ell_2 := \frac{30}{M}$. The beam is modelled by partial differential equations based on beam theory from Timoshenko and Gere ([15]), which then are solved using the Finite Element Method. A more profound explanation of the modelling can be found in [14].

The problem to be solved is then

$$\begin{array}{ll} \underset{\mathbf{x}=(\mathbf{x}_1,\ldots,\mathbf{x}_K)}{\text{minimize}} & f(\mathbf{x}) \\ \text{subject to} & \mathbf{x}_i = (x_i^h, x_i^w) \in \{(h,w) \in \mathbb{R}^2 : (h,w) \in X^h \times X^w\} \quad i = 1,\ldots,K \end{array} \tag{9}$$

where f measures the deflection, and $X^h := \{h \in \mathbb{R} : h = 450 + k\ell_1, k = 0, \ldots, , M\}$ and $X^w := \{w \in \mathbb{R} : w = 20 + k\ell_2, k = 0, \ldots, M\}$ are the sets of the heights and the widths respectively.

4.2 Benchmarking of algorithms

In order to assess and measure the performance of the algorithms we use the techniques introduced by MorÃ© and Wild ([16]), and Dolan and MorÃ© ([17]), namely *data profiles* and *performance profiles*. Let $\mathcal{A}$ be the set of the algorithms. The idea is to gather data by applying each algorithm $a \in \mathcal{A}$ on a large set of test problems $\mathcal{P}$. In our case the set of test problems consists of an equal amount of instances of the beam problem and the artificial problem, where each problem will be generated with a random number of variables as well as a random number of designs for the variables. For a problem $p \in \mathcal{P}$ and a algorithm $a \in \mathcal{A}$, the performance metric $t_{p,a}$, defined as the number of function evaluations required to satisfy a convergence test, will be used to construct the profiles. The following convergence test, suggested by MorÃ© and Wild ([16]), is used; a feasible design $\mathbf{z}$ satisfies the convergence test if

$$f(\mathbf{z}^0) - f(\mathbf{z}) \geq (1-\tau)(f(\mathbf{z}^0) - f_{min}). \tag{10}$$

Here $\mathbf{z}^0$ is the starting point for the problem, f_{min} is computed for each problem as the smallest objective value found using any of the algorithms, and $\tau \in [0,1]$ is the tolerance parameter representing the desired decrease from the starting objective value $f(\mathbf{z}^0)$. Hence, the algorithm is said to converge at the feasible point $\mathbf{z}$ if the reduction $f(\mathbf{z}^0) - f(\mathbf{z})$ is at least $(1-\tau)$ times the reduction $f(\mathbf{z}^0) - f_{min}$. By convention, if the algorithm a fails to satisfy the convergence test for problem p, then $t_{p,a} = \infty$.

The performance profile, originally introduced by Dolan and MorÃ© ([17]), is obtained from the performance metric by computing the the performance ratio, that is for problem p and algortihm a

$$r_{p,a} = \frac{t_{p,a}}{\min\{t_{p,a} : a \in \mathcal{A}\}}.$$

The performance profile for algorithm a is then defined as

$$\rho_a(\alpha) = \frac{1}{|\mathcal{P}|}\text{size}\{p \in \mathcal{P} : r_{p,a} \leq \alpha\}. \tag{11}$$

Thus, the performance profile is the cumulative distribution function for the ratio $r_{p,a}$. Plotting these functions for all algorithms provides information about the relative performance of each algorithm and the probability for each algorithm to satisfy the convergence test. Note that evaluating the function $\rho_a(\alpha)$ at $\alpha = 1$ gives the percentage of problems on which algorithm a converges with the least amount of function evaluations compared to the other algorithms. Additionally, the value of α when the function first attains the value 1, yields that the algorithm never requires more than α times the best algorithm's number of function evaluations in order to converge. However, the performance profile does not provide sufficient information when the measure of interest is expensive function evaluations, for instance if there is an upper limit on the number function evaluations allowed. In order to account for this, we use the data profile as additional way to measure performance. The data profile for algorithm a is defined as

$$d_a(\beta) = \frac{1}{|\mathcal{P}|}\text{size}\{p \in \mathcal{P} : t_{p,a} \leq \beta\}, \tag{12}$$

that is, the percentage of problems that satisfy the convergence test in equation (10) within β function evaluations.

Additionally, examples of plots of the objective value against the number of function evaluation will be provided in order to illustrate the trajectory leading to the optimal value found.

5 Implementation in Matlab

In this section a thorough description of how the algorithms, the test problems, and the benchmarking were implemented in Matlab is presented. The implementation was done in Matlab R2015b and carried out on a computer equipped with Intel Core i5 2.7 GHz and 8 GB RAM, running macOS Sierra. The code implementing the algorithms can be found in Appendix B.

5.1 Local search algorithm

The implementation of the local search algorithm was decomposed into two parts; a main file consisting of the basic steps of the algorithm and a function finding the neighborhood. We have chosen to use the categorical 1-neighborhood and the discrete 3-neighborhood. These are henceforth referred to as the categorical and the discrete neighborhood. The decomposition allowed us to alter between the discrete and categorical neighborhood without affecting the remaining steps of the algorithm. A description of the MATLAB implementation of the main steps of the pseudo code introduced in Section 3.2 follows.

The first step of the local search algorithm, namely choosing a starting point, was made randomly generating a uniformly distributed random integer between 1 and the number of feasible designs for each of the variables.

The second step of the algorithm was to find the neighborhood for a given problem and a configuration. In opposition to traversing the feasible set and successively add those configurations that satisfies the definition, the neighborhood functions were implemented to construct the set of configurations differing in the given number of elements.

Lastly, the termination criterion was implemented as a boolean variable, keeping track of whether there was a configuration with a lower objective value in the neighborhood or not. Depending on the value of this boolean variable either a new search iteration is performed resulting in a new better configuration or the local search algorithm (Algorithm 1) terminates and returns the optimal configuration found.

In addition, we created a function in MATLAB which returns the optimal point and has the starting point as an input parameter. Also, this function includes a counter, allowing us to keep track of how many function evaluations it takes to reach a minimum. Further, we implemented the local search algorithm with both categorical and discrete neighborhood.

The MATLAB code finding the neighborhoods can be found in Appendix B.1 and the MATLAB code performing the local search algorithm can be found in Appendix B.2.

5.2 Global search algorithm

The implementation of global search algorithm is partly based on the local search algorithm. In addition to the local search algorithm described in the previous section, the global search algorithm contains two additional steps: computing the weight vector and the new starting point. The global search algorithm was only implemented with the discrete neighborhood and the MATLAB code can be found in Appendix B.3.

The size of the starting set was chosen to be three points with the aim to increase the probability of finding a successful local minimum while still limiting the total number of function evaluations until termination. These starting points were chosen randomly from the feasible set. Then, three local searches were performed producing a vector of three local minima. The vector of local minimum is sorted and then each local minimum is assigned its corresponding weight. In order to obtain the new starting point we multiply the weight vector with the vector of local minima. However, it is not assured that this new point is a feasible configuration. Thus, we round this new point according to equation (7). Starting from this new point, we perform a local search yielding a new local minimum. Lastly, we compare the four local minima found and return the one with the lowest objective value.

5.3 Genetic algorithm

The genetic algorithm can be varied in many different ways, and what follows is a description of the operators we used in our implementation of the algorithm. For full MATLAB code see Appendix B.4.

For our encoding scheme in the artificial problem with m variables a configuration $\mathbf{x}^j = (x_1^j, \dots, x_m^j) \in X$ was encoded as a chromosome $\mathbf{c}^j = (c_1^j, \dots, c_m^j)$ with the genes $c_1^j = x_1^j, \dots, c_m^j = x_m^j$. For the beam problem, a configuration with K beam sections was encoded as a chromosome $\mathbf{c}^j = (c_1^j, \dots, c_{2K}^j)$ with genes $c_1^j = x_1^w$, $c_2^j = x_1^h, \dots, c_{2K-1}^j = x_K^w$, $c_{2K}^j = x_K^h$, where x_k^w is the width of beam section k, and x_k^h is the height of beam section k. These types of encoding schemes are not very efficient for problems with few variables ([12]), but we used it for simplicity. Population sizes anywhere between 30 and 1000 are common ([12]), but to keep down the number of function evaluations, we used 20 individuals.

Since both the beam problem and the artificial problem are minimization problems, the fitness values of the individuals were assigned as the additive inverse of the objective value. The reason why we did not use multiplicative inverse is that the objective function in the artificial problem may take on negative values. For example, individuals with objective values -2 and 2 would then have been assigned fitness values $-\frac{1}{2}$ and $\frac{1}{2}$ respectively, which does not comply with the convention that better individuals should be assigned higher fitness values.

The best set of parameter values depend on the problem and are not known beforehand. We therefore chose parameters quite arbitrarily within reasonable intervals that usually perform alright. The tournament selection parameter should be in the interval $(0.5, 1)$, typically around 0.7-0.8 ([12]). We used tournament selection with a tournament size of two randomly selected individuals, of which the one with higher fitness was selected with probability 0.7.

Crossover was applied using one randomly generated crossover point. The same point was used for both individuals since the encoding used in both problems required a fixed chromosome length. The crossover probability can be anywhere in the interval $[0, 1]$ and varies a lot between problems, and we set it to 0.6 in our implementation.

We used two kinds of mutations, regular mutation where a gene is replaced by a randomly generated one, and a variation of the so called *creep mutation*. We chose for mutation to happen to each gene with probability $\frac{1}{m}$, where m is the number of genes, since can be shown that a mutation probability somewhere around $p_{mut} = \frac{1}{m}$ usually works well ([12]). Once it had been determined that mutation should take place at all the probability of regular mutation was 0.5 and creep mutation 0.5 since we did not know which type of mutation would work best.

Instead of generating the value of new gene randomly, the creep mutation in the beam problem either increased or decreased the value of the gene by the smallest amount allowed, or did nothing to the gene, each with an equal probability of $\frac{1}{3}$. In practice this means that a dimension of a beam section would be increased or decreased one step. For the artificial problem creep mutation meant randomizing a new integer within the interval of $\pm 5\%$ of the allowed variable range surrounding the old gene. Looking at values close to the old one could be thought of as fine tuning the solution. However, for the creep mutation to be able to do any fine tuning in the case of a categorical optimization problem, there needs to be an underlying structure to the problem. Without some sort of underlying structure, there would be no reason to believe that a value close the old gene would result in a better solution than a value far away.

One copy of the best individual was inserted into the new generation in the elitism step each iteration, and it always replaced the first individual in the generated population.

5.4 Benchmarking

In order to benchmark the algorithms we used 100 instances of the test problems, 50 beam problems and 50 artificial problems. The beam problems were generated with a random number of segments between 2 and 6 and a random number of feasible designs common for all segments between 10 and 30. The artificial problems were generated according to section 4.1.

The set of algorithms consists of the local search algorithm with categorical neighborhood LS-C, the local search with discrete neighborhood LS-D, the global search with discrete neighborhood GS-D, and the genetic algorithm GA. Hence we have that $\mathcal{A} = \{\text{LS-C, LS-D, GS-D, GA}\}$.

Before we applied the algorithms to these problems we computed the convergence criterion, that is the right hand side of inequality (10). That requires a starting point, a known approximate optimal solution to the problem and a value of the tolerance parameter τ. To make the performance comparison as fair as possible we let the genetic algorithm generate an initial population and then randomize the starting point for the rest of the algorithms from this population. Further, the best approximate solutions were found in different ways depending on the type of problem. The beam problems have a known optimal solution, namely the maximal height and width of each segment. However, the optimal solutions to the artificial problems are not known. We did empirical tests, comparing the different algorithms, which yielded that the local search with categorical neighborhood was the best algorithm to apply the artificial problem in order to get an approximate optimal solution. The tolerance parameter, representing the sought decrease from the starting objective value, was chosen to be $\tau = 0.1$ and $\tau = 0.01$.

Then, equipped with starting points and a convergence criterion, we applied the algorithms to each of the problems and recorded the number of objective function evaluations until the convergence criterion was satisfied. If the algorithms failed to satisfy the convergence criterion before terminating, the performance metric was set to infinity. Note that for the genetic algorithm, the maximum number of function evaluation is predetermined by the population size times the number of generations. We set the number of generations to 300 yielding a maximum number of evaluations to 6000. If, however, the genetic algorithm did not converge within this number of function evaluations the performance metric were set to infinity. This resulted in a matrix with the dimension $100 \cdot |\mathcal{A}|$. Further, we created the vectors α and β with 100 uniformly spaced points in $[1, 40]$ and $[1, 6000]$ respectively. Then, after using equation (11) and (12) with the acquired matrix and α and β we plot the data profile and the performance profile.

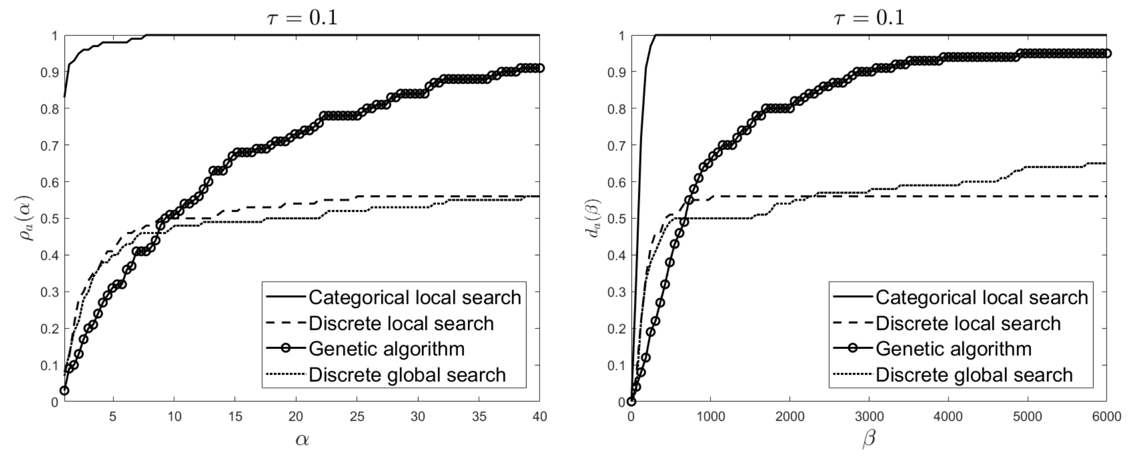
6 Results

This section presents the benchmarking results of the algorithms introduced in Section 3. First, the data and performance profiles will be illustrated alongside some underlying numerical values for the two choices of the tolerance parameter τ. Then, for two specific examples of the beam and artificial problem, an illustration of the objective value against the number of function evaluations is shown.

In Figure 5 the performance profile and data profile for the algorithms are shown with tolerance parameter $\tau = 0.1$. The main numerical results are illustrated below in Table 1.

Table 1: The number of problems where the algorithm $a \in \mathcal{A}$ converges according to (10) with $\tau = 0.1$, $\eta_a = \text{card}\{p \in \mathcal{P} : t_{p,a} < \infty\}$, and the number of problems for which the algorithm converges in the least amount of function evaluations compared to the other algorithms, $\rho_a(1)$.

	LS - C		**LS - D**		**GA**		**GS - D**	
	η_a	$\rho_a(1)$	η_a	$\rho_a(1)$	η_a	$\rho_a(1)$	η_a	$\rho_a(1)$
Beam	50	34	50	7	45	3	50	6
Artificial	50	49	6	1	50	0	22	0
Total	100	83	56	8	95	3	72	6



(a) Performance profile (b) Data profile

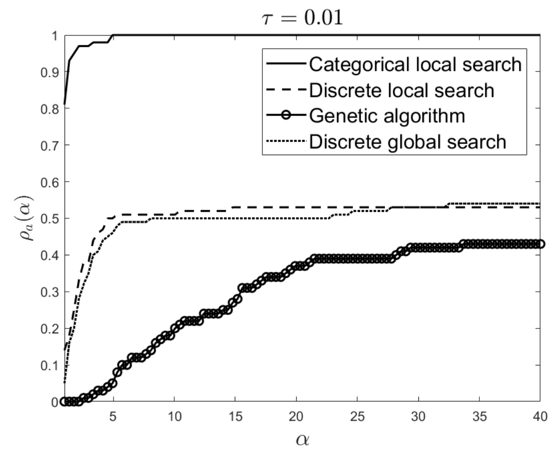
Figure 5: Data profile $d_a(\alpha)$ and performance profile $\rho_a(\beta)$ for the algorithms applied to 50 instances of beam problems and 50 instances of artificial problems with $\tau = 0.1$.

For this randomized set of 50 instances each of the test problems, the categorical local search was the only algorithm that converged in all problem instances. In addition, it converges in the least number of function evaluations in 83% of the problems. In contrast, the genetic algorithm converges for 95% of the problem instances, but only in 3 cases with the least number of function evaluations. The discrete local search converges in all instances of the beam problem, but only in 3 instances of the artificial problem. In comparison, the discrete global search also converges in all instances of the beam problem, however, it converges in 22 cases of the artificial problem. Additionally, since $\rho_a(\alpha) = 1$ when $\alpha \geq 8$ for the categorical local search algorithm, this algorithm never requires more than 8 times the number of function evaluations needed for the best performing algorithm.

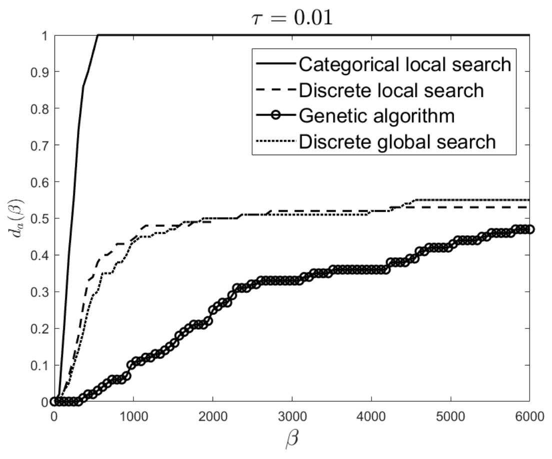
In Figure 6 the performance profile and data profile for the algorithms are shown with the tolerance parameter $\tau = 0.01$. The main numerical results are illustrated below in Table 2.

Table 2: The number of problems where the algorithm $a \in \mathcal{A}$ converges according to (10) with $\tau = 0.01$, $\eta_a = \text{card}\{p \in \mathcal{P} : t_{p,a} < \infty\}$, and the number of problems for which the algorithm converges in the fewest function evaluations compared to the other algorithms, $\rho_a(1)$.

	LS - C		**LS - D**		**GA**		**GS - D**	
	η_a	$\rho_a(1)$	η_a	$\rho_a(1)$	η_a	$\rho_a(1)$	η_a	$\rho_a(1)$
Beam	50	32	50	13	4	0	50	5
Artificial	50	49	3	1	43	0	8	0
Total	100	81	53	14	47	0	58	5



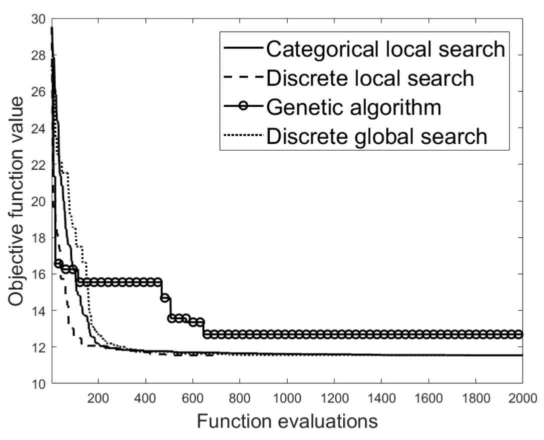
(*a*) Performance profile



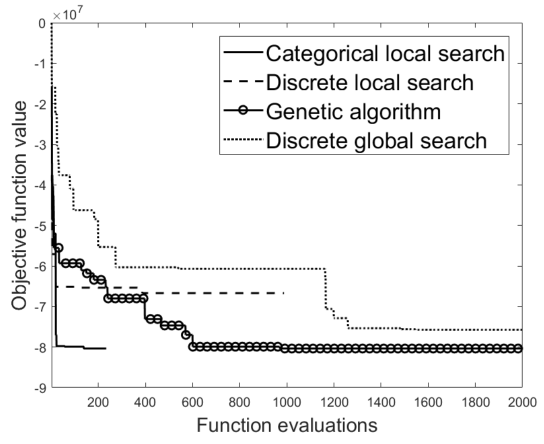
(*b*) Data profile

Figure 6: Data profile $d_a(\alpha)$ and performance profile $\rho_a(\beta)$ for the algorithms applied to 50 instances of the beam problems and 50 instances of the artificial problems with $\tau = 0.01$.

With this stricter tolerance parameter, the categorical local search is still the only algorithm that converges in all problem instances. The number of problem instances where it converges in the least number of function evaluation is about the same, this time 81%. Similarly to the case $\tau = 0.1$, both the discrete local search and the discrete global search algorithms converge in all instances of the beam problem. The number of instances of the artificial problem in which they converge is smaller though. Finally, the number of instances where the genetic algorithm converges is reduced by over 50% from 95 to 47, converging only in 4 instances of the beam problem. In addition, it is never the first algorithm to converge. This time we see that for the categorical local search, $\rho_a(\alpha) = 1$ for $\alpha \geq 5$, hence it never requires more than 5 times the number of function evaluations needed for the best performing algorithm before it converges.



(*a*) Beam problem with five segments



(*b*) Artificial problem with five variables

Figure 7: Example of a plot of the objective value against the number of function evaluations for all algorithms on the two test problems.

In Figure 7, an example of the algorithms solving an instance of the beam problem (a) and an instance of the artificial problem (b) is shown. The objective value is plotted against the number of function evaluations. In the beam problem, the genetic terminates at 6000 function evaluations and the global search algorithm terminates at 9640 function evaluations. The corresponding numbers of function evaluations for the artificial problem are 6000 for the genetic algorithm and 4541 for the global search algorithm. However, the graphs are cut at 2000 function evaluations since neither the genetic algorithm nor the global search algorithm descend in any of the problem instances after that. Similarly to the results from the profiles, both the local searches and the global search performs better than the genetic algorithm on the beam problem. In the artificial problem, the local search with categorical neighborhood and the genetic algorithm terminates at the same configuration.

7 Discussion

Initially we will discuss the results of the numerical tests of the different algorithms. Then the possible implications for TyreOpt research project will be laid out.

7.1 Algorithms

The results show that according to the data and performance profiles, the categorical local search algorithm performs the best out of the tested algorithms. We will now discuss the credibility of this result.

Firstly, although we have used a total of 100 different test problems, it may not necessarily be the case that these are representative enough. It could have been valuable if there were more test problems of different character, rather than only two. Secondly, while the choice of f_{min} for the beam problems was given since the global optimum was known, it could have been done differently in the case of the artificial problems. In order to find a better approximate optimal solution, one could have let all algorithms solve the problem and take the best one. However, the choice of using the categorical local search was made because it is faster. It could have been interesting to only include test problems with known optimal solutions a priori, in order to see how many times the global optimum was found. However, the convergence test served as an adequate substitute.

Although the categorical neighborhood performs better than the discrete one in general, it is worth noting that the discrete local search converged first in 13 instances of the beam problem. Since the beam problem has an underlying structure, it may be beneficent to use a discrete neighborhood in situations where this is the case. As the categorical local search converges first in 49 out of the 50 instances of the artificial problem (see Table 2) with the stricter convergence criterion, the categorical neighborhood seems to be the appropriate choice when there is no underlying structure.

The fact that the genetic algorithm took such a downfall when the tolerance parameter was changed from 0.1 to 0.01 to get a stricter convergence criterion is a sign that it manages to find approximate solutions, but not exact and efficient ones, where the latter is measured in number of function evaluations. Improvements of the genetic algorithm could be made by adjusting the parameter values depending on the problem at hand. Choosing different implementations of the operators, such as roulette-wheel selection or two-point crossover could also possibly improve the performance of the algorithm. Another possible enhancement could be to choose a different encoding scheme that produces longer chromosomes. However, all of these possible improvements are problem dependent, and it is difficult to know beforehand what modifications will improve the algorithm when solving a specific problem.

In the global search algorithm one can think of several ways to improve the algorithm. One example of a way to improve the algorithm is to change how to choose the starting points in S. In our version of the algorithm, we simply chose them randomly out of the set of feasible configurations but with an a priori known underlying structure of the problem the choosing procedure could for example be based on some computations. One thing to compute could be distance between points and it may be a good idea to have a large distance between starting points since we want to minimize the risk that the different local searches make the same function evaluations. If we have a discrete problem a suggestion of a different starting point set would be the *corner points*, or some of the corner points. The set of corner points X^c of a configuration set X, with n elements in each configuration, is simply defined as $X^c := \{\mathbf{x} \in X \mid \mathbf{x} + \mathbf{d} \in X \implies \mathbf{x} - \mathbf{d} \notin X,\ \mathbf{d} \in \{\pm \mathbf{e}^i \mid i = 1, \dots, n\}\}$, where $\mathbf{e}^i$ is the vector with $e^i_i = 1$ and $e^i_j = 0\ \ \forall i \neq j$. Some other examples of improving the global search algorithm could be to change the construction of the new starting point **y** or having several new starting points instead of one.

It is noteworthy that an algorithm such as the local search yields better results than the genetic algorithm, while being very simplistic in nature based on basic optimization theory. Further, there does not exist many algorithms for categorical optimization, and much research has not been done within this topic.

7.2 TyreOpt

The photos of tires provided by Volvo GTT can be found in Appendix A. These photos were supposed to be used to parametrize the tire patterns. When a suitable parametrization is found, the tire tread patterns can be used as additional decision variables in the tires selection problem being solved with TyreOpt. However, in our opinion the supplied photos were not sufficiently well taken. The photographs of the tires were not taken straight from the front but slightly from the side making it complicated to do further numerical analysis. If the photos were taken from the front we suggest three possible parameterizations using image analysis. Firstly, one could define variables based on the percentage of the photo of the tire that consists of groove. Secondly, one could use the number of different directions of the groove patterns. Lastly, one could extract variables based on the direction in which the majority of the grooves were directed. The tread pattern then results in several additional decision variables to be added to the vehicle dynamics models developed within TyreOpt project. For all the parametrizations suggested both the neighborhoods definitions introduced in Section 2 can be used allowing the optimization methodology developed within TyreOpt research project to be applied for the extended tires selection without any further changes.

8 Conclusion

In this project we have implemented three algorithms for solving categorical optimization problems. Two of these algorithms depend on neighborhoods that we have introduced. We have found that with a suitable neighborhood and a relatively simple algorithm, it is possible to outperform genetic algorithms, which have previously been used frequently within categorical problems. In addition, we have improved the local search algorithm by expanding it into a global search algorithm. Nonetheless, we believe that the global search algorithm could become more sophisticated after further development.

With regards to the TyreOpt project, we found that the photos of the tires provided were not sufficiently useful. However, if the photos are taken as we have recommended, the tread pattern can be directly used as an additional set of decision variables in the tire selection problem. For this purpose, we have suggested three parametrizations of the tire treads.

We encourage further use and development of neighborhood-based algorithms in favor of the genetic algorithms, as they provide an opportunity to deepen the understanding of categorical optimization.

References

[1] Carlson SE, Shonkwilerm R, Ingrim E. Comparison of three non-derivative optimization methods with a genetic algorithm for component selection. Journal of Engineering Design. 1994;5(4):367–378.

[2] Nedělková Z. Optimization of truck tyres selection. Chalmers University of Technology. Department of Mathematical Sciences; 2018. PhD thesis.

[3] Rothlauf F. Design of Modern Heuristics: Principles and Application. 1st ed. Springer Publishing Company, Incorporated; 2011.

[4] Andréasson N, Evgrafov A, Patriksson M, Gustavsson E, Nedělková Z, Sou KC, et al. An introduction to continuous optimization. Lund, Sweden: Studentlitteratur; 2016.

[5] Parker RG, Rardin RL. Discrete Optimization. San Diego, CA, USA: Academic Press Professional, Inc.; 1988.

[6] Brown DR, Hwang KY. Solving fixed configuration problems with genetic search. Research in Engineering Design. 1993;5(2):80–87.

[7] Carlson-Skalak S, White MD, Teng Y. Using an evolutionary algorithm for catalog design. Research in Engineering Design. 1998;10(2):63–83.

[8] Lindroth P, Patriksson M. Pure categorical optimization: a global descent approach. Chalmers University of Technology, University of Gothenburg, Department of Mathematical Sciences; 2011. Technical report.

[9] Ng CK, Li D, Zhang LS. Discrete global descent method for discrete global optimization and nonlinear integer programming. Journal of Global Optimization. 2007;37(3):357–379.

[10] Fuchs M, Neumaier A. Discrete search in design optimization. In: Complex Systems Design & Management. Springer; 2010. p. 113–122.

[11] Carlson SE. Genetic algorithm attributes for component selection. Research in Engineering Design. 1996;8(1):33–51.

[12] Wahde M. Biologically inspired optimization methods: an introduction. Ashurst Lodge, Ashurst, Southampton, UK: WIT press; 2008.

[13] Holland JH. Adaptation in Natural and Artificial Systems. University of Michigan Press; 1975.

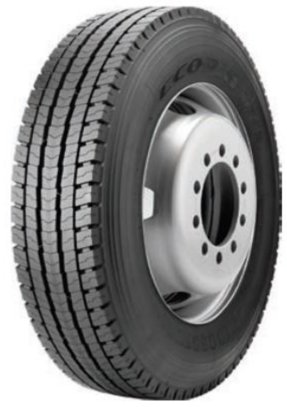
[14] Thanedar P, Vanderplaats G. Survey of discrete variable optimization for structural design. Journal of Structural Engineering. 1995;121(2):301–306.

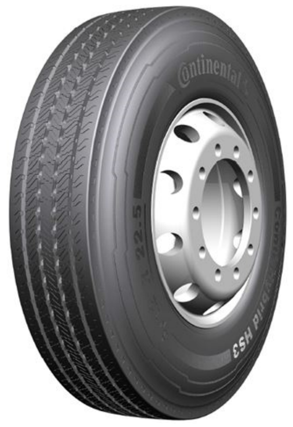
[15] Timoshenko S, Gere JM. Mechanics of Materials. Boston, MA, US: Van Nostrand Reinhold Co.; 1972.

[16] Moré JJ, Wild SM. Benchmarking derivative-free optimization algorithms. SIAM Journal on Optimization. 2009;20(1):172–191.

[17] Dolan ED, Moré JJ. Benchmarking optimization software with performance profiles. Mathematical programming. 2002;91(2):201–213.

A Photos of tires


(a) Tire 1


(b) Tire 2

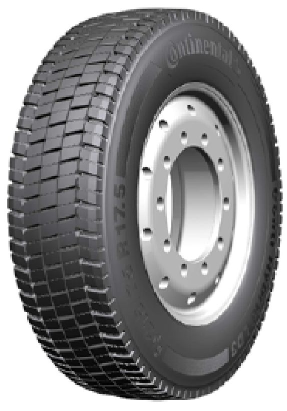

(c) Tire 3

Figure 8: Photos of tires supplied by Volvo GTT.

B Matlab code

B.1 Neighborhoods

B.1.1 Categorical

```
function neighborhood = catNeigh(currentSetting,nFeasDes)
% Initiate
neighborhood=[];
nOfVar=length(currentSetting);

% Loop through each variable
for i = 1 : nOfVar
    % Configurations that differ in variable i
    neighTemporary = zeros(nFeasDes(i),nOfVar);
    for k = 1:nFeasDes(i)
        newSetting = currentSetting;
        newSetting(i) = k;
        neighTemporary(k,:) = newSetting;
    end
    % Concatenate neighbourhood
    neighborhood = [neighborhood ; neighTemporary];
end
% Randomly order the neighborhood
neighborhood=neighborhood(randperm(length(neighborhood)),:);
end
```

B.1.2 Discrete

```
function neighborhood = discreteNeighForReport(setting,nOfVar,nFeasDes)

% Configurations that differs
sizeNeigh1= length(setting)*4+4*sum(length(setting)-1);
neigh1 = zeros(sizeNeigh1,length(setting));
k=1;
for i = 1 : nOfVar
    for l = 1:nOfVar
        if l~=i
            % Both up
            neigh1(k,:) = setting;
            neigh1(k,i) = setting(i) +1;
            neigh1(k,l) = setting(l) +1;

            % Both down
            neigh1(k+1,:) = setting;
            neigh1(k+1,i) = setting(i) -1;
            neigh1(k+1,l) = setting(l) -1;

            % First down, second up
            neigh1(k+2,:) = setting;
            neigh1(k+2,i) = setting(i) -1 ;
            neigh1(k+2,l) = setting(l) +1;

            % First up, second down
            neigh1(k+3,:) = setting;
            neigh1(k+3,i) = setting(i) +1 ;
            neigh1(k+3,l) = setting(l) -1;
            k = k + 4;
        end
    end
end

% Configuretaions wich differ in 1-3 indexes in one variable
k = 1;
for i = 1 : nOfVar

    % Two index up
    neigh2(k,:) = setting;
    neigh2(k,i) = setting(i) + 2;

    % One index up
    neigh2(k+1,:) = setting;
    neigh2(k+1,i) = setting(i) + 1;

    % One index down
    neigh2(k+2,:) = setting;
    neigh2(k+2,i) = setting(i) -1;

    % Two index down
    neigh2(k+3,:) = setting;
    neigh2(k+3,i) = setting(i) -2;

    % Three index up
    neigh2(k+4,:) = setting;
    neigh2(k+4,i) = setting(i) + 3;

    % Three index down
    neigh2(k+5,:) = setting;
    neigh2(k+5,i) = setting(i) - 3;

    k = k + 6;

end

% Configurations which differ in two index in one variable, and one index
% in another variable
neigh3Size = sum(1:(nOfVar-1));
neigh3=zeros(neigh3Size,nOfVar);
k=1;
for i = 1 : nOfVar

    % Change one index two up or two down
    neighTmp = setting;
    neighTmp(i) = setting(i) + 2;

    neighTmp2 = setting;
    neighTmp2(i) = setting(i) - 2;

    for index = 1 : nOfVar
        if index ~= i
            % Config. where first change was 2 up
            neightempo1 = neighTmp;
            neightempo2 = neighTmp;

            % Config. where first change was 2 down
            neightempo3 = neighTmp2;
            neightempo4 = neighTmp2;

            % Change one index one up or down
            neightempo1(index) = neighTmp(index)+1;
            neightempo2(index) = neighTmp(index)-1;

            % Change one index one up or down
            neightempo3(index) = neighTmp2(index)+1;
            neightempo4(index) = neighTmp2(index)-1;

            % Add new configurations to the neighbourhood
            neigh3(k,:) = neightempo1;
            neigh3(k+1,:) = neightempo2;
            neigh3(k+2,:) = neightempo3;
            neigh3(k+3,:) = neightempo4;
            k=k+4;
        end
    end
end


% Configurations that differ in 3 variables by 1 index
neigh4=[];
for i = 1 : nOfVar
    % Change one index up
    neightmp1 = setting;
    neightmp1(i) = setting(i) + 1;

    % Change one index down
    neighTmp2 = setting;
    neighTmp2(i) = setting(i) - 1;

    for o = 1:nOfVar
        if o ~=i
            % Change index:
            % First was up. Change second to second up
            neightmp11 = neightmp1;
            neightmp11(o) = setting(o) + 1;

            % First was up. Change second to second down
            neightmp12 = neightmp1;
            neightmp12(o) = setting(o) - 1;

            % First was down. Change second to second up
            neightmp21 = neighTmp2;
            neightmp21(o) = setting(o) + 1;

            % First was down. Change second to second down
            neightmp22 = neighTmp2;
            neightmp22(o) = setting(o) - 1;

            for p = 1:nOfVar
```

```
                if p ~= o && p ~=i

                    % First up. Second second up, third up
                    neightmp111 = neightmp11;
                    neightmp111(p) = setting(p) + 1;

                    % First up. Second second up, third down
                    neightmp112 = neightmp11;
                    neightmp112(p) = setting(p) - 1;

                    % First up. Second second down, third up
                    neightmp121 = neightmp12;
                    neightmp121(p) = setting(p) + 1;

                    % First up. Second second down, third down
                    neightmp122 = neightmp12;
                    neightmp122(p) = setting(p) - 1;

                    % First down. Second second up, third up
                    neightmp211 = neightmp21;
                    neightmp211(p) = setting(p) + 1;

                    % First down. Second second up, third down
                    neightmp212 = neightmp21;
                    neightmp212(p) = setting(p) - 1;

                    % First down. Second second down, third up
                    neightmp221 = neightmp22;
                    neightmp221(p) = setting(p) + 1;

                    % First down. Second second down, third down
                    neightmp222 = neightmp22;
                    neightmp222(p) = setting(p) - 1;

                    % Concatenate
                    neigh4=[neigh1;neightmp111;neightmp112;...
                        neightmp121;neightmp122;neightmp211;...
                        neightmp212;neightmp221;neightmp222];
                end
            end
        end
    end
end

% Concatenate all different neighborhoods
neigh = [neigh1;neigh2;neigh3;neigh4];


% Find index of configurations that are infeasible
index = [];
for n = 1 : nOfVar
    indexTmp1 = find(neigh(:,n) <= 0);
    indexTmp2 = find(neigh(:,n) > nFeasDes(n));
    index = [index;indexTmp1;indexTmp2];
end
```

```
% Remove duplicate index
index = unique(index);

% Remove configurations with the found index
for o = length(index) :-1 :1
    neigh(index(o),:) =[];
end

% Randomly order the neigh
neigh = neigh(randperm(length(neigh)),:);

% Return neighborhood
neighborhood = neigh;


end
```

B.2 Local search

B.2.1 Artificial problem

```
% Generate random starting point
for i = 1 : numberOfVariables
    currentSetting(i) = randi(nFeasDes(i));
end

currentBestObjValue = ObjFunc(currentSetting,zMapping);

% Local Search
while true
    abort = true;

    % Discrete neigh
    neigh = discreteNeigh(currentSetting,numberOfVariables,nFeasDes);

    %Categorical neigh
    % neigh = catNeigh(currentSetting,nFeasDes);

    for i = 1 :length(neigh)
        candidate = ObjFunc(neigh(i,:),zMapping);
        if currentBestObjValue > candidate
            currentBestObjValue = candidate;
            currentSetting = neigh(i,:);
            abort = false;
            break
        end
    end

    if abort == true
        break
    end
end
```

B.2.2 Beam problem

```
% Generate random starting point
for i = 1:numBlocks
```

```
    currentSetting(:,i) = Z(:,randi(N));
end

% Objective value of startingpoint
currentBestObjValue=trueObj(currentSetting(:));

while true
    abort = true;
    % Categorical neigh
    %neigh=catNeigh(currentSetting,numDesigns,numBlocks);

    % Discrete neigh
    neigh = discreteNeigh(currentSetting, numDesigns);

    % Loop through neighbourhood
    for i = 1:length(neigh)/2
        candidateSetting =  neigh(2*i-1:2*i,:);
        candidate = objectiveFunction(candidateSetting(:));
        if candidate<currentBestObjValue % Check if new point is better
            currentBestObjValue=candidate; % Update to better point
            currentSetting = candidateSetting; % Update best objective value
            abort = false; % Set termination critera to false
            break
        end
    end

    % Check if termination criteria is true
    if abort == true
        break
    end
end
```

B.3 Global search

B.3.1 Artificial Problem

```
function [counter,objVector,globMin]=globSearch(nOfStartingPoints,...
    nOfVar,nFeasDes,zMapping)

% Generate starting points
for i=1:nOfStartingPoints
    for j=1:nOfVar
        % Random integer in [0, number of designs]
        startpoints(i,j)=randi(nFeasDes(j));
    end
end

% Initiate
nextPoints=zeros(nOfStartingPoints,nOfVar);
counter=0;
objVector=[];

% Loop through starting points
for j=1:nOfStartingPoints
    % Perform local search from starting point
    [counter,objVector,nextPoints(j,:)]=locSearchWithCounter(...
        startpoints(j,:),nOfVar,nFeasDes,zMapping,counter,objVector);
```

```
end

% Objective value of the new acquired points
objValuesOfNewPoints=zeros(nOfStartingPoints,1);
for k=1:nOfStartingPoints
    objValuesOfNewPoints(k)=objectiveFunction(nextPoints(k,:),zMapping);
end

% Sort objective values descending
[~, indexes] = sort(objValuesOfNewPoints,'descend');

% Find weights
weight=(1/sum(indexes))*indexes;

% Construct new starting point
y=zeros(nOfVar,1);
for m=1:nOfVar
    for n=1:nOfStartingPoints
        y(m)=y(m)+weight(n)*nextPoints(n,m);
    end
end

% Round to closest points in order to make it feasible
y=round(y);

% Perform local search from the new point
[counter,objVector,y]=locSearchWithCounter(y,nOfVar,nFeasDes,...
    zMapping,counter,objVector);

% Compare all local min. and return the lowest
if objectiveFunction(y,zMapping)>objectiveFunction(nextPoints(...
        indexes(end),:),zMapping)
    globMin=objectiveFunction(nextPoints(indexes(end),:),zMapping);
else
    globMin=objectiveFunction(y,zMapping);
end
end
```

B.3.2 Beam Problem

```
function [counter,objVector,globMin]=beamProblemGlobalSearch(...
numBlocks,numDesigns,Z,N)
% Number of starting points
nOfStartingPoints=3;

% Construct startpoints
startpoints=cell(nOfStartingPoints,1);
for i=1:nOfStartingPoints
    for j = 1:numBlocks
        % Random design
        startpoints{i}(:,j) = Z(:,randi(N));
    end
end

% Initiate
nextPoints=cell(nOfStartingPoints,1);
```

```
counter=0;
objVector=[];

% Loop through starting points
for j=1:nOfStartingPoints
    % Perform local search from starting point
    [counter,objVector,nextPoints{j}]=beamLocalSearchWithCounter(...
        startpoints{j},numBlocks,numDesigns,counter,objVector);
end

% Objective value of the new acquired points
objValues=zeros(numBlocks,1);
for k=1:numBlocks
    objValues(k)=objectiveFunction(nextPoints{k});
end

% Sort objective values descending
[~, indexes] = sort(objValues,'descend'); % descend

% Find weights
weight=(1/sum(indexes))*indexes;

% Construct new starting point
y=zeros(2,numBlocks);
for r=1:2 % rows in setting
    for m=1:numBlocks
        for n=1:numBlocks
            y(r,m)=y(r,m)+weight(n)*nextPoints{n}(r,m);
        end
    end
end

% Round to closest points in order to make it feasible
newB=B;
newH=H;
for j=1:numBlocks
    newB=sort([newB y(1,j)]);
    y(1,j)=B(find(newB==y(1,j),1)); % Width
    newB=B;
    newH=sort([newH y(2,j)]);
    y(2,j)=H(find(newH==y(2,j),1)); % Height
    newH=H;
end

% Perform local search from the new point
[counter,objVector,y]=beamLocalSearchWithCounter(y,numBlocks,...
    numDesigns,counter,objVector);

% Compare all local min. and return the lowest
if objectiveFunction(y)>objectiveFunction(nextPoints{indexes(numBlocks)})
    globmin=objectiveFunction(nextPoints{indexes(numBlocks)});
else
    globmin=objectiveFunction(y);
end
end
```

B.4 Genetic algorithm

B.4.1 Artificial problem

Main file

```
main_artificial; % Generates problem (nFeasDes and zMapping)

populationSize = 20;
numberOfGenes = length(nFeasDes); % number of variables/dimensions
crossoverProbability = 0.60;
mutationProbability = 1/numberOfGenes;
creepMutationProbability = 0.50;
tournamentSelectionParameter = 0.70;
variableRange = nFeasDes; % different range for different variables
numberOfGenerations = 20;
numberOfEvaluations = populationSize*numberOfGenerations;
fitness = zeros(populationSize,1);

population = InitializePopulation(populationSize,numberOfGenes,variableRange);

for iGeneration = 1:numberOfGenerations

    % Evaluation of individuals
    maximumFitness = -Inf;
    xBest = zeros(1,2);
    bestIndividualIndex = 0;

    for i = 1:populationSize

        chromosome = population(i,:);
        x = chromosome;
        fitness(i) = EvaluateIndividual(x,zMapping);

        if (fitness(i) > maximumFitness)
            maximumFitness = fitness(i);
            bestIndividualIndex = i;
            xBest = x;
        end
    end

    tempPopulation = population;

    % Selection of individuals
    for i = 1:2:populationSize

        i1 = TournamentSelect(fitness,tournamentSelectionParameter);
        i2 = TournamentSelect(fitness,tournamentSelectionParameter);
        chromosome1 = population(i1,:);
        chromosome2 = population(i2,:);

        % Crossover individuals
        r = rand;
        if (r < crossoverProbability)
            newChromosomePair = Cross(chromosome1,chromosome2);
            tempPopulation(i,:) = newChromosomePair(1,:);
            tempPopulation(i+1,:) = newChromosomePair(2,:);
        else
            tempPopulation(i,:) = chromosome1;
            tempPopulation(i+1,:) = chromosome2;
        end
    end

    % Mutate individuals
    for i = 1:populationSize
        originalChromosome = tempPopulation(i,:);
        mutatedChromosome = Mutate(originalChromosome,mutationProbability,...
          creepMutationProbability,variableRange);
        tempPopulation(i,:) = mutatedChromosome;
    end

    % Elitism (best individual is cloned into next generation)
    tempPopulation(1,:) = population(bestIndividualIndex,:);
    population = tempPopulation;

end
```

Function initializing population

```
function population = InitializePopulation(populationSize,...
    numberOfGenes,variableRange)

for i = 1:populationSize
    for j = 1:numberOfGenes
        population(i,j) = randi(variableRange(j));
    end
end
```

Function evaluating an individual

```
function f = EvaluateIndividual(x,zMapping)

objectiveValue = ObjFunc(x,zMapping); % Calls objective function
f = -objectiveValue;
```

Tournament selection function

```
function iSelected = TournamentSelect(fitness,tournamentSelectionParameter)

populationSize = size(fitness,1);

iTmp1 = 1 + fix(rand*populationSize);
iTmp2 = 1 + fix(rand*populationSize);

r = rand;
if (r < tournamentSelectionParameter)
    if (fitness(iTmp1) > fitness(iTmp2))
        iSelected = iTmp1;
    else
        iSelected = iTmp2;
    end
else
    if (fitness(iTmp1) > fitness(iTmp2))
        iSelected = iTmp2;
    else
        iSelected = iTmp1;
    end
end
```

Crossover function

```
function newChromosomePair = Cross(chromosome1,chromosome2)

nGenes = size(chromosome1,2);
crossoverPoint = 1 + fix(rand*(nGenes-1));

for j = 1:nGenes
    if (j < crossoverPoint)
        newChromosomePair(1,j) = chromosome1(j);
        newChromosomePair(2,j) = chromosome2(j);
    else
        newChromosomePair(1,j) = chromosome2(j);
        newChromosomePair(2,j) = chromosome1(j);
    end
end
```

Mutation function

```
function mutatedChromosome = Mutate(chromosome,mutationProbability,...
    creepMutationProbability,variableRange)

nGenes = size(chromosome,2);
mutatedChromosome = chromosome;

for j = 1:nGenes

    r = rand;
    if (r < mutationProbability) % mutation happens

        q = rand;
        if (q < creepMutationProbability) % creep mutation happens

            fivePercent = round(variableRange(j)/20);
            newGene = chromosome(j) - fivePercent + randi([0 2*fivePercent]);

            % check variables within range
            newGene = max(newGene, 1);
            newGene = min(newGene, variableRange(j));

            mutatedChromosome(j) = newGene;
        else % regular mutation happens
            mutatedChromosome(j) = randi(variableRange(j));
        end
    end
end
```

B.4.2 Beam problem

Main file

```
numBlocks = 6; % number of blocks of the beam
```

```
numDesigns = 30; % number of designs for each block
volFac = 0;

populationSize = 20;
numberOfGenes = numBlocks*2; % number of variables/dimensions
crossoverProbability = 0.6;
mutationProbability = 1/numberOfGenes;
creepMutationProbability = 0.5;
tournamentSelectionParameter = 0.7;
variableRange = [20 50; 450 600]; % [Bmin Bmax; Hmin Hmax]
numberOfGenerations = 30;
numberOfEvaluations = populationSize*numberOfGenerations;
fitness = zeros(populationSize,1);

population = InitializePopulation(populationSize, numberOfGenes,...
  variableRange, numDesigns);

for iGeneration = 1:numberOfGenerations

    % Evaluation of individuals
    maximumFitness = 0.0;
    xBest = zeros(1,2);
    bestIndividualIndex = 0;

    for i = 1:populationSize
        chromosome = population(i,:);
        x = chromosome;
        fitness(i) = EvaluateIndividual(x,volFac);

        if (fitness(i) > maximumFitness)
            maximumFitness = fitness(i);
            bestIndividualIndex = i;
            xBest = x;
        end
    end

    tempPopulation = population;

    % Selection of individuals
    for i = 1:2:populationSize

        i1 = TournamentSelect(fitness,tournamentSelectionParameter);
        i2 = TournamentSelect(fitness,tournamentSelectionParameter);
        chromosome1 = population(i1,:);
        chromosome2 = population(i2,:);

        % Cross over indiviuals
        r = rand;
        if (r < crossoverProbability)
            newChromosomePair = Cross(chromosome1,chromosome2);
            tempPopulation(i,:) = newChromosomePair(1,:);
            tempPopulation(i+1,:) = newChromosomePair(2,:);
        else
            tempPopulation(i,:) = chromosome1;
            tempPopulation(i+1,:) = chromosome2;
        end
    end

    % Mutate individuals
    for i = 1:populationSize
        originalChromosome = tempPopulation(i,:);
        mutatedChromosome = Mutate(originalChromosome,mutationProbability,...
          creepMutationProbability,variableRange,numDesigns);
        tempPopulation(i,:) = mutatedChromosome;
    end

    % Elitism (best individual is cloned into next generation)
    tempPopulation(1,:) = population(bestIndividualIndex,:);
    population = tempPopulation;
end
```

Function initializing population

```
function population = InitializePopulation(populationSize, numberOfGenes,...
  variableRange, numDesigns)

for i = 1:populationSize
    for j = 1:numberOfGenes

        if mod(j,2) == 1 % odd dimension, i.e. W (width)
          population(i,j) = min(variableRange(1,:)) + ...
            (randi(numDesigns)-1)*diff(variableRange(1,:))/(numDesigns-1);

        else % even dimension, i.e. H (height)
          population(i,j) = min(variableRange(2,:)) + ...
            (randi(numDesigns)-1)*diff(variableRange(2,:))/(numDesigns-1);
        end
    end
end
```

Function evaluating an individual

```
function f = EvaluateIndividual(x,volFac)

objectiveValue = objfunc(x(:),volFac); % Calls objective function
f = -objectiveValue;
```

Tournament selection function and crossover function

The tournament selection and crossover functions were the same for the artificial problem and the beam problem. See section B.4.1 for code.

Mutation function

```
function mutatedChromosome = Mutate(chromosome,mutationProbability,...
  creepMutationProbability,variableRange,numDesigns)

nGenes = size(chromosome,2);
mutatedChromosome = chromosome;

for j = 1:nGenes

    r = rand;
    if (r < mutationProbability) % mutation happens

        q = rand;
        if (q < creepMutationProbability) % creep mutation happens

            if mod(j,2) == 1 % i.e B
                mutatedChromosome(j) = chromosome(j) + ...
                  (randi([0 2])-1)*diff(variableRange(1,:))/(numDesigns-1);
                % Check variables within range
                if mutatedChromosome(j) > max(variableRange(1,:))
                    mutatedChromosome(j) = max(variableRange(1,:));
                elseif mutatedChromosome(j) < min(variableRange(1,:))
                    mutatedChromosome(j) = min(variableRange(1,:));
                end

            else % i.e. H
                mutatedChromosome(j) = chromosome(j) + ...
                  (randi([0 2])-1)*diff(variableRange(2,:))/(numDesigns-1);
                % Check variables within range
                if mutatedChromosome(j) > max(variableRange(2,:))
                    mutatedChromosome(j) = max(variableRange(2,:));
                elseif mutatedChromosome(j) < min(variableRange(2,:))
                    mutatedChromosome(j) = min(variableRange(2,:));
                end
            end

        else % regular mutation happens
            if mod(j,2) == 1 % i.e. B
                mutatedChromosome(j) = min(variableRange(1,:)) + ...
                  (randi(numDesigns)-1)*diff(variableRange(1,:))/(numDesigns-1);

            else % i.e. H
                mutatedChromosome(j) = min(variableRange(2,:)) + ...
                  (randi(numDesigns)-1)*diff(variableRange(2,:))/(numDesigns-1);
            end
        end
    end
end
```

C Summary in Swedish

C.1 Inledning

Det här kandidatarbetet grundar sig på forskningsprojektet TyreOpt utfört av Chalmers tekniska högskola och Göteborgs universitet i samarbete med Volvo Group Trucks Technology. Det huvudsakliga syftet med forskningsprojektet var att minska bränsleförbrukningen hos lastbilar genom att optimera valet av däck. Däcken betraktas som diskreta variabler utan naturlig ordning. Detta innebär att det ej går att ordna dem, såsom man kan ordna tal efter deras storlek. Variabler av detta slag kallas kategoriska. Avsaknaden av naturlig ordning gör att befintliga metoder för kontinuerliga och diskreta optimeringsproblem inte kan användas i samma utsträckning. Därför studerar vi inledningsvis grundläggande koncept såsom avstånd och omgivning för kategoriska variabler. Därefter ämnar vi undersöka befintliga algoritmer samt utveckla egna varianter av dem. Avslutningsvis kommer algoritmernas prestanda att utvärderas.

Utöver detta förs en en diskussion kring vilka förbättringar som kan göras av modeller utvecklade i TyreOpt. I synnerhet analyserar vi bilder på däckmönster i syfte att skapa nya kategoriska variabler som kan inkluderas och förbättra modellerna.

C.2 Optimeringslära

Optimeringslära är den gren inom matematiken som söker metoder för att hitta den bästa lösningen till ett givet problem. För att åstadkomma detta ställs en matematisk modell av problemet upp bestående av en målfunktion som beskriver hur väl en uppsättning val, så kallade beslutsvariabler, löser problemet. En sådan uppsättning kallas en lösning. Mängden av lösningar som är tillåtna specifieras genom bivillkor. Målet är att minimera eller maximera målfunktionen, det vill säga hitta en tillåten lösning som ger ett lägre respektive högre värde än alla andra lösningar på målfunktionen, ett så kallat globalt optimum. Ett maximeringsproblem kan alltid uttryckas som ett minimeringsproblem genom att ändra tecken på målfunktionen. Av denna anledning använder vi fortsättningsvis orden optimum och minimum synonymt.

Det är inte alltid ett globalt optimum existerar, till exempel om målfunktionen är obegränsad. Däremot kan en lösning ha lägst eller högst målfunktionsvärde i en omgivning kring sig själv. En sådan lösning kallas ett lokalt optimum. Omgivningen till lösningen kan väljas på olika sätt beroende på vilka lösningar som ska ligga nära varandra i lösningsrummet, och bestäms vid uppställningen av problemet. Till exempel brukar omgivningen tas som en Euklidisk boll om lösningsrummet är $\mathbb{R}^n$.

Ett diskret optimeringsproblem är ett problem vars lösningsrum är en delmängd av $\mathbb{Z}^n$. Jämfört med problem vars lösningsrum är en delmängd till $\mathbb{R}^n$, är dessa svårare att lösa. Detta beror på att antalet tillåtna lösningar växer snabbt när antalet beslutsvariabler ökar. Detta utesluter möjligheten att evaluera målfunktionen för samtliga lösningar för att hitta det största eller lägsta värdet. Givetvis existerar inte denna möjlighet för $\mathbb{R}^n$ heller, som har oändligt många punkter. Däremot finns verktyg från matematisk analys tillgängliga, som gör det möjligt att ta genvägar och undvika evaluering av samtliga lösningar. Sådana tekniker har inte utvecklats för diskreta problem med samma framgång, och diskreta algoritmer är därav mer krävande.

C.3 Kategorisk optimering

I vissa problem utgörs beslutsvariablerna inte av tal. Ett exempel är om man ska minimera värmeflödet ut ur ett hus genom att välja ett lämpligt material. De olika tillåtna materialen, som kallas inställningar, utgör då lösningsrummet. Till skillnad från tal finns det inget naturligt sätt att jämföra olika material såsom det är möjligt säga att ett tal är större än ett annat. Med andra ord är dessa variabler kategoriska, och problemet är ett så kallat kategoriskt optimeringsproblem.

Ett kategoriskt optimeringsproblem kan omvandlas till ett diskret optimeringsproblem genom att numrera de olika inställningarna. Antag att vi har ett problem där vi behöver välja n komponenter, och för komponent j finns det m_j möjliga inställningar, där $j = 1, \ldots, n$. Vi kan då tilldela varje inställning ett tal $k \in \{1, \ldots, m_j\} =: X^j$, och således kan lösningsrummet uttryckas som $X = X^1 \times \cdots \times X^j$. Varje lösning kan då representeras som $\mathbf{x} = (x_1, \ldots, x_n)$ där $x_j \in X^j$. Följaktligen kan lösningsrummet X ses som en delmängd till $\mathbb{Z}^n$. Eftersom inställningar kan tilldelas tal på ett godtyckligt sätt, kommer ordningen som skapas inte nödvändigtvis representera en underliggande struktur i problemet.

Ett sätt att definiera avståndet mellan två lösningar är genom Hamming-avståndet. Hamming-avståndet mellan två lösningar $\mathbf{x}$ och $\mathbf{y}$ definieras som antalet komponenter som skiljer dem åt, det vill säga storleken på mängden $\{i \in \{1, \ldots, n\} : x_i \neq y_i\}$. Utifrån Hamming-avståndet kan vi definiera en kategorisk omgivning till $\mathbf{x}$ som de lösningar som skiljer sig från $\mathbf{x}$ på högst en komponent. Denna omgivning används för att definiera ett lokalt kategoriskt optimum. Det går också att definera en diskret omgivning, eftersom det kategoriska problemet omvandlats till ett diskret problem. Den diskreta omgivning är en diskretiserad version av en Euklidisk boll.

C.4 Algoritmer för kategoriska optimeringsproblem

För att lösa kategoriska optimeringsproblem använde vi oss av tre olika algoritmer. Den första algoritmen är en så kallad lokal sökmetod. Givet att vi har definierat en omgivning och bestämt en startpunkt, så evalueras målfunktionen för varje lösning i startpunktens omgivning tills att ett lägre värde på målfunktionen än målfunktionens värde i startpunkten hittats. När en sådan lösning påträffats är nästa steg i algoritmen att upprepa proceduren fast för omgivningen till den förbättrade lösningen. Detta upprepas fram tills att vi har löpt igenom en hel omgivning utan att hitta ett lägre målfunktionsvärde. Då terminerar algoritmen och returnerar startpunkten från den senaste iterationen.

Med hjälp av den lokala sökmetoden utformar vi en global sökmetod, som ämnar öka chansen att hitta ett globalt optimum till problemet. Denna metod inleds med att slumpmässigt generera tre olika startpunkter, och utifrån dessa tillämpa den lokala sökmetoden för att få tre lokala minima. Därefter konstrueras en ny startpunkt genom att vikta de tre erhållna lokala minima utifrån hur lågt deras målfunktionsvärde är med förhoppningen att hitta en bättre startpunkt. Denna konstruktion garanterar ej att den nya startpunkten faktiskt är en tillåten lösning. För att komma runt detta använder vi en avrundning. Sedan utför vi en lokalsökning från den nya förvärvade startpunkten och erhåller ytterligare ett lokalt minimum. Slutligen returnerar vi det minsta av de fyra lokala minima vi hittat.

Den tredje algoritmen är en så kallad genetisk algoritm. En egenskap hos den genetiska algoritmen är att den inte kräver en omgivningsdefinition, vilket gör den till en alternativ metod att tillämpa på kategoriska optimeringsproblem. Algoritmen går ut på att man kodar konfigurationer eller punkter i sökrummet som strängar av tal, vilka kallas kromosomer. Varje tal i en kromosom kallas för en gen. Det första man gör är att slumpmässigt generera en population bestående av ett antal kromosomer. Därefter evalueras alla kromosomer i målfunktionen och tilldelas ett så kallat lämplighetsvärde, där ett högre värde innebär att kromosomen motsvarar en bättre lösning till optimeringsproblemet.

Baserat på kromosomernas lämplighetsvärden så görs därefter en dragning av individer som får föröka sig till nästa generation av populationen. Dragningen görs med återläggning. Kromosomer med högre lämplighetsvärde har större sannolikhet att väljas. Kromosomerna väljs ut i par om två, och för varje par är det en viss sannolikhet att något som kallas överkorsning äger rum. Det innebär att kromosomerna byter vissa gener sinsemellan.

När urvalet och överkorsningen är avklarad så utför man något som kallas mutation. Det innebär att varje gen i varje kromosom i den nya populationen har en viss sannolikhet att mutera, det vill säga bytas ut mot en helt ny. Slutligen så kopierar man över den bästa kromosomen från den gamla till den nya populationen, för att försäkra sig om att inte förlora den, vilket kallas för elitism. Hela processen upprepas antingen i ett förbestämt antal generationer, eller tills något konvergenskriterium är uppnått.

C.5 Bedömningsmetodik

För att utvärdera hur väl algoritmerna presterade så valde vi ut två kategoriska testproblem som algoritmerna fick försöka lösa. Det första testproblemet kallar vi för det artificiella problemet. En instans av det artificiella problemet genereras genom att slumpmässigt välja både storleken på och innehållet i de matriser och vektorer som utgör målfunktionen. Det andra testproblemet, som vi kallar för balkproblemet, består i att optimera dimensionerna på en utskjutande balk så att böjningen minimeras när en kraft appliceras längst ut på balken. Balken består av flera delsektioner där varje sektion ska tilldelas en viss höjd och bredd. En instans av balkproblemet genereras genom att slumpmässigt välja både antalet delsektioner i balken och vilka dimensioner på delsektionerna som är tillåtna.

Som måttstock för hur väl algoritmerna löste problemen så användes så kallade data- och prestandaprofiler. Vi lät algoritmerna lösa en stor mängd testproblem och registrerade antalet målfunktionsevalueringar som utförts fram tills att ett specifikt konvergenstest är uppfyllt. I vårt fall bestod mängden av testproblem av lika många instanser av balkproblem och artificiella problem, där instanserna genererats slumpmässigt med avseende på antalet variabler samt inställningar. Data- och prestandaprofilerna ger sedan information om den relativa prestandan för varje algoritm och andelen av de genererade testproblemen som varje algoritm uppfyller konvergenstestet för.

C.6 Resultat och slutsats

Data- och prestandaprofilerna visade att lokalsökningsalgoritmen med kategorisk omgivning i konvergerade samtliga testproblem. Dessutom konvergerade den först i 83% av testproblemen. Jämförelsevis konvergerade den genetiska algoritmen i 95% av testproblemen, men endast i 3% av samtliga testproblemen konvergerade den först. Både lokalsöknings- och globalsökningsalgoritmen med den diskreta omgivningen konvergerade i alla balkproblem, men presterade betydligt sämre på de artificella problemen.

Resultaten visar att av de algoritmer vi har testat, baserat på våra testproblem, så presterar lokalsökningsalgoritmen med kategorisk omgivning bäst. Värt att nämna är att lokalsökningsalgoritmen med diskret omgivning är klart sämre än den kategoriska. Vi ser alltså att inte bara valet av algoritm spelar roll utan även valet av omgivning. Vidare såg vi att den genetiska algoritmen krävde relativt många evalueringar av målfunktionen för att hitta bra lösningar.

Vi har utvecklat lokalsökningsalgoritmen genom att utvidga den till en globalsökningsalgoritm. Den presterade något bättre än lokalsökningsalgoritmen med diskret omgivning. Emellertid tror vi att globalsökningsalgoritmen har potential att förbättras ytterligare.

Beträffande forskningsprojektet TyreOpt så kom vi fram till att de foton av däck som vi försågs med inte var tillräckligt vältagna. Fotona togs ej rakt framifrån, vilket försvårade vidare analys av däckmönstren. Därför föreslog vi hur de borde tas, och rekommenderade utifrån detta tre olika parametriseringar, till exempel andelen av däcket som utgörs av mönsterskåror.