Department of Mathematics
Chair of Applied Geometry and Discrete Mathematics

Applying the RNBI Method for Multicriteria Optimization Problems to a Manpower Planning Problem

Interdisciplinary Project (IDP) by Radu Vintan

Examiner:	Dr. René Brandenberg
Advisor:	Wolfgang F. Riedl
Submission Date:	March 2, 2022

Abstract

Multicriteria Optimisation is concerned with problems that require optimising multiple objective functions simultaneously. Usually, improving one objective function can lead to the deterioration of others. Hence, there is a need for finding suitable trade-offs between conflicting objectives. *Nondominated* solutions, i.e. solutions for which we cannot improve any objective without worsening others, are particularly interesting in this context. The *RNBI* method is an algorithm which computes a suitable discrete representation of the (possibly infinite) set of nondominated solutions. It can be used for multicriteria optimisation problems with linear constraints and objective functions. For this Interdisciplinary Project (IDP), we implemented and used RNBI to find nondominated solutions for a practical *Manpowering* problem.

Zusammenfassung

Multikriterielle Optimierung befasst sich mit Problemen, bei denen mehrere Zielfunktionen gleichzeitig optimiert werden müssen. In der Regel kann die Verbesserung einer Zielfunktion zu einer Verschlechterung anderer Zielfunktionen führen. Daher müssen geeignete Kompromisse zwischen konkurrierenden Zielen gefunden werden. *Nichtdominierte* Lösungen, d.h. Lösungen, bei denen wir kein Ziel verbessern können, ohne andere zu verschlechtern, sind in diesem Zusammenhang besonders interessant. Die *RNBI*-Methode ist ein Algorithmus, der eine geeignete diskrete Darstellung der (möglicherweise unendlichen) Menge der nichtdominierten Lösungen berechnet. Sie kann für multikriterielle Optimierungsprobleme mit linearen Nebenbedingungen und Zielfunktionen verwendet werden. In diesem Interdisziplinären Projekt (IDP) haben wir RNBI implementiert und verwendet, um nichtdominierte Lösungen für ein praktisches *Manpowering* Problem zu finden.

Contents

Chapter 1

Introduction

In multicriteria (or multi-objective) optimization we are given $p \geq 2$ objective functions which we need to optimize (per convention *minimize*) simultaneously. Usually, improving some objectives leads to the deterioration of others. In particular, this implies that there does not exist a solution that is optimal for all objective functions at the same time. In this context, one needs to settle for solutions which are *nondominated*, i.e. solutions for which we cannot improve any objective without worsening others. Multicriteria optimization has been used to model many practical problems arising in diverse fields of science, including operations research, economics and logistics.

For this Interdisciplinary Project (IDP), we will focus on *Multiple Objective Linear Programming* problems (MOLP), i.e. multicriteria optimisation problems where both the constraints and the objective functions can be formulated using linear expressions on a finite set of predefined variables [Ehr05]. We note that this is a generalization of *Linear Programming* (LP). This particular form does not reduce the usefulness excessively. Many applications propose real-life problems that can be formulated using just linear constraints and objectives. One example is given by the *Manpowering problems*, which model the need of a company to hire new employees over a long time span while ensuring that certain strategical objectives are met.

What is expected from an ideal MOLP solver? The set Y_N of nondominated solutions may be infinite, and hence we need to settle for a discretized representation $R \subset Y_N$ of this set. A MOLP solver outputs such an R. Then, a *Decision Maker* (DM) inspects R and uses their intimate knowledge of the problem to choose a solution from it. What makes a representation R suitable for a human DM? Preferably, for every nondominated solution y, we ought to find some representative $r \in R$ which is similar to y. This is called *small representation error*. In this way the DM is "informed" that a solution like y is possible. Moreover, we would like to avoid receiving representatives r_1, r_2 which are too similar to each other, as this creates redundancy. This is called *high uniformity*. Finally, the size of R should be reasonably small, to enable the DM to look at all proposed solutions.

Two of the first algorithms designed to find a suitable discrete representation R for a MOLP have been the *Global Shooting Method*, proposed by Benson and Sayin

[BS97], and the *Normal Boundary Intersection Method* (NBI), proposed by Das and Dennis [DD98]. Unfortunately, none of these two algorithms guarantees both small representation error and high uniformity. In 2007, Shao and Ehrgott [SE07] combined the ideas from the Global Shooting and NBI methods to design the *Revised Normal Boundary Intersection Method* (RNBI), which has provably small representation error and high uniformity. This makes RNBI an excellent algorithm to use for solving MOLPs. Since 2007, more efficient implementations of RNBI have been proposed, for example, using Column Generation techniques [LER17; Rua20].

For the purpose of this IDP, we have implemented the (classical) RNBI method and applied it to a concrete Manpowering problem, which was given to us as a Black Box. We visualize the results and show them to the DM, who can then choose a suitable solution. The outline of this document is as follows: In Chapter 2 we present the theory behind MOLP solvers and, in particular, the Global Shooting, NBI and RNBI methods. In Chapter 3 we discuss our implementation of RNBI and the results we obtained. We also show an extension of RNBI which allows the user to have more flexibility in choosing where specifically to search for nondominated solutions. In Chapter 4 we discuss potential continuations or improvements of our work which could be interesting for a future project.

Chapter 2

Multi-objective Linear Programming

A *Multiple Objective Linear Programming* (MOLP) problem is given by:

$$\min\{Cx : x \in X\} \tag{2.1}$$

where $C \in \mathbb{R}^{p \times n}$ is a matrix describing the p different objective functions we aim to minimize simultaneously. $X \subseteq \mathbb{R}^n$ is hereby a nonempty bounded polyhedron which describes the *feasible set in solution space*. The condition $x \in X$ can be enforced using just linear constraints. Note that, if $p = 1$, then (2.1) is just a LP, and can be solved using well known algorithms such as the Simplex method. Therefore, we will assume $p \geq 2$ and be aware that there likely exists no solution which is optimal for all objectives at the same time.

Let $Y := \{Cx : x \in X\} \subseteq \mathbb{R}^p$ be the *feasible set in objective space*. Y is also a bounded polyhedron [Roc70], and the relation between solution space and objective space is shown in Figure 2.1. We will call a solution $y \in Y$ *nondominated* if there exists no $y' \in Y \setminus \{y\}$, s.t. $y' \leq y$, i.e. y' improves some objective without worsening any other. Figure 2.2 shows the difference between dominated and nondominated points for a particular case where $p = 2$.

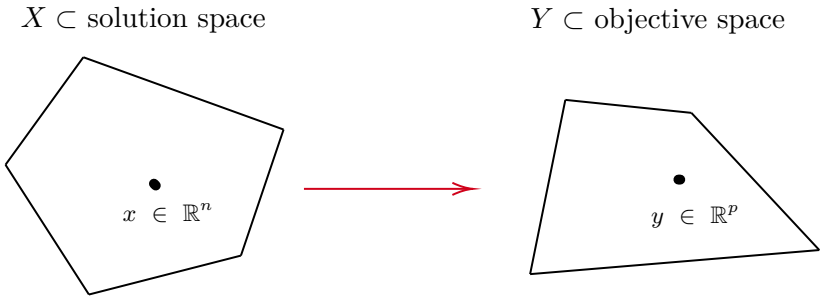

Figure 2.1: Visualizing the feasible sets in solution space and objective space respectively.

Let $Y_N \subset Y$ be the subset of nondominated solutions. Usually, Y_N is infinite, and cannot be outputted by an MOLP solver. We instead wish to find a suitable discrete representation $R \subset Y_N$ that a *Decision Maker* (DM) can use to choose a suitable

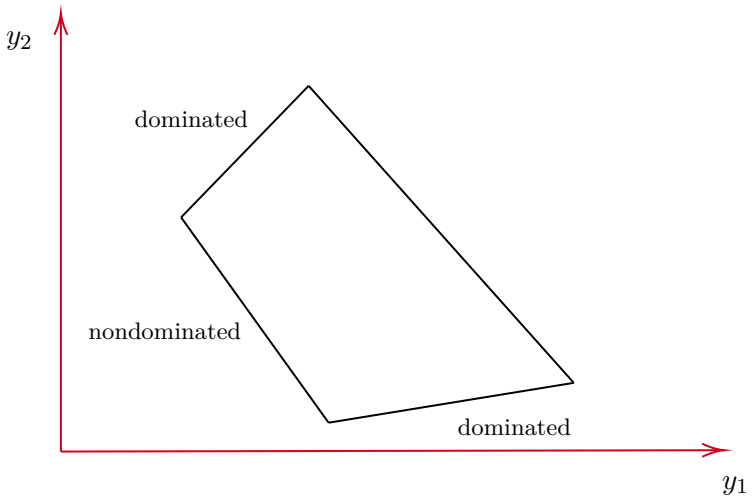

Figure 2.2: Visualizing the difference between dominated and nondominated solutions.

solution. The DM is a person with enough intimate knowledge of the concrete MOLP problem that they can properly decide on a solution if they are given a relatively small set R of potential solutions. The quality of R is evaluated using two quantities, besides relatively small size:

Definition 2.1
Given a discrete representation R of Y_N, the quantity:

$$\varepsilon := \sup\{||r - y|| : y \in Y_N, r \in R\} \tag{2.2}$$

is called the *representation error* of R. We wish for ε to be as small as possible, because this guarantees that the DM has access to a diverse set of solutions which accurately capture the reality of Y_N.

Definition 2.2
Given a discrete representation R of Y_N, the quantity:

$$\delta := \inf\{||r - r'|| : r, r' \in R, r \neq r'\} \tag{2.3}$$

is called the *uniformity* of R. We wish for δ to be as big as possible, because it ensures that we have little redundancy in our solution.

The previous two definitions are also displayed graphically in Figure 2.3. An ideal MOLP solver should have provably low representation error and high uniformity. In the next sections, we will present three classical algorithms that produce a discrete

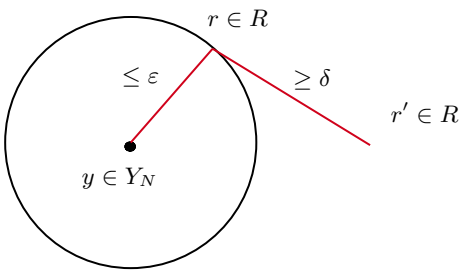

Figure 2.3: Showing the two relevant conditions for a discrete representation R of Y_N

representation R of Y_N. These are the Global Shooting Method, the Normal Boundary Intersection Method (NBI), and the Revised Normal Boundary Intersection Method (RNBI).

2.1 Global Shooting Method

The Global Shooting Method has been introduced by Benson and Sayin in [BS97]. It is quite similar to the RNBI method, and we will therefore explain it in more detail. Note that the feasible set in objective space Y is bounded and compact. This implies that each separate objective function has a finite maximum inside Y. This allows the following:

Definition 2.3
We indicate by y_i the i-th coordinate of a vector $y \in \mathbb{R}^p$, $i \in [p]$. Let

$$y_i^{AI} := \max\{y_i : y \in Y\} \in \mathbb{R} \tag{2.4}$$

Then y^{AI}, i.e. the vector containing the individual maxima of the objectives, is called the *anti ideal* point. Note that in general we might have $y^{AI} \notin Y$. We also note that one can similarly define the *ideal* point y^I, which contains the individual minima of the objective functions.

Computing the individual maxima can be done using an LP solver. This is because the condition $y \in Y$, equivalent to $y = Cx,\ x \in X$ can be written with only linear constraints. Intuitively, y^{AI} lies "above" the bounded polyhedron Y. In the next step of the algorithm, we will construct a simplex S that "embraces" Y:

Theorem 2.4 ([BS97])
Let $\beta := \min\{\langle e, y\rangle : y \in Y\}$, where $e \in \mathbb{R}^p$ is a vector of ones. We define $p+1$ points

$v^k \in \mathbb{R}^p$, $k \in \{0, 1, \dots, p\}$. *Hereby* $v^0 = y^{AI}$ *and for* $k \in [p]$ *we define:*

$$v_i^k := \begin{cases} y_i^{AI}, & \text{if } i \neq k \\ \beta + y_k^{AI} - \langle e, y^{AI} \rangle & \text{if } i = k \end{cases} \tag{2.5}$$

Then the convex hull S *of the* $p+1$ *defined points is a* p*-dimensional simplex which contains* Y*. Moreover, the subsimplex* $\hat{S}$ *of* S *given by the convex hull of* $v^1, \dots, v^p$ *is a supporting hyperplane of* Y_N *with normal* e*.*

Intuitively, the subsimplex $\hat{S}$ lies right below the polyhedron Y. Computing β can be done using an LP solver. This also enables us to compute the coordinates of the points v^k which define S (and $\hat{S}$). The next step of the algorithm consists of sampling points on $\hat{S}$. We then connect all these sampled points with the anti ideal point y^{AI}. Figure 2.4 shows the result of this procedure in the particular case $p = 2$. The points on $\hat{S}$ are sampled *uniformly*. We provide a definition because the same sampling will be used for RNBI:

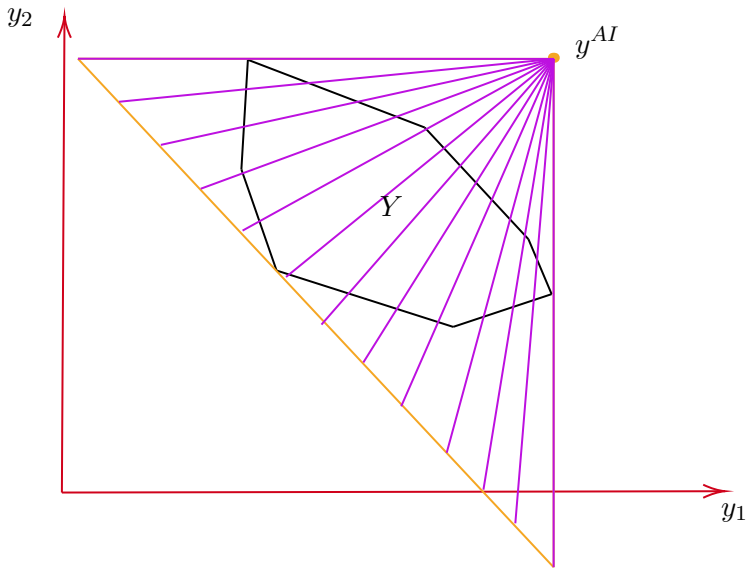

Figure 2.4: We sample points on the subsimplex $\hat{S}$, i.e. the line opposite to y^{AI}, and connect all of them with the anti ideal point. In this way we find points on the boundary of Y.

Definition 2.5
Let $m \in \mathbb{Z}_{\geq 1}$ be a parameter, and $\eta := 1/m$. The *uniform sampling* on the subsimplex

$\hat{S}$ with parameter m samples following points:

$$\text{Unif}(\hat{S}, m) := \left\{ \sum_{k=1}^{p} \alpha_k v^k : \alpha_k \text{ multiple of } \eta, \ \sum_{k=1}^{p} \alpha_k = 1 \right\} \tag{2.6}$$

That is, the sampling takes all convex combinations where the coefficients are restricted to be multiples of $\eta := 1/m$ (including 0 and 1). The greater m is, the more points we sample.

As explained, we send rays from each of the sampled points $q \in \text{Unif}(\hat{S}, m)$ towards y^{AI} and look at the (first) intersection point of this ray with Y. This can be done by solving the LP: $\min\{t : t \in \mathbb{R}_{\geq 0}, q + t(y^{AI} - q) \in Y\}$ and using the obtained optimum t^* to compute the intersection point $q_{hit} := q + t^*(y^{AI} - q) \in Y$. Note that q_{hit} may be dominated. This is indeed the case for some of the points we find in Figure 2.4. One can algorithmically detect such points and eliminate them, but we defer this discussion to Section 2.3. Eventually, after eliminating those points, we receive a discrete representation R of Y_N. Does R have provably small representation error and high uniformity? Unfortunately, only the former property has been proven, while the latter does not seem to hold (see Section 3.3 in [SE07]). This means that the Global Shooting Method might give us many redundant solutions. We continue by discussing the next algorithm, which will fix this at the cost of losing the former property.

2.2 Normal Boundary Intersection Method (NBI)

This NBI Method has been introduced by Das and Dennis in [DD98]. In contrast to the Global Shooting Method, NBI begins by considering the individual minima of the objective functions. However, it does not use the *ideal* point y^I, i.e. the vector that contains these individual minima, but the convex hull of the points on Y where the individual minima are attained. The following definition makes this idea precise:

Definition 2.6
Again, y_i indicates the i-th coordinate of $y \in \mathbb{R}^p$. Let

$$y_i^* := \arg\min\{y_i : y \in Y\} \in \mathbb{R}^p \tag{2.7}$$

The points $y_i^*, i \in [p]$ define the *convex hull of the individual minima* (CHIM).

The NBI method samples points uniformly on the CHIM. Then it sends rays perpendicular to the CHIM "downwards" towards Y and calculates the intersection points. This is displayed in Figure 2.5. We skip any technical details here because the RNBI method uses a different approach. Like for the Global Shooting Method, the

points we obtain by this procedure might be dominated. Again, we defer to Section 2.3 for a discussion on how this can be addressed. After eliminating the dominated points, we receive a discrete representation R of Y_N. This time, R will have provably high uniformity, but unfortunately no provably small representation error. In fact, Shao and Ehrgott provide an example where the representation error is very high (Section 3.5 in [SE07]). Hence, the NBI Method is in some way the dual of the Global Shooting Method. What one method does better the other does worse. In the next section we will see how ideas from both approaches have been combined to alleviate these issues.

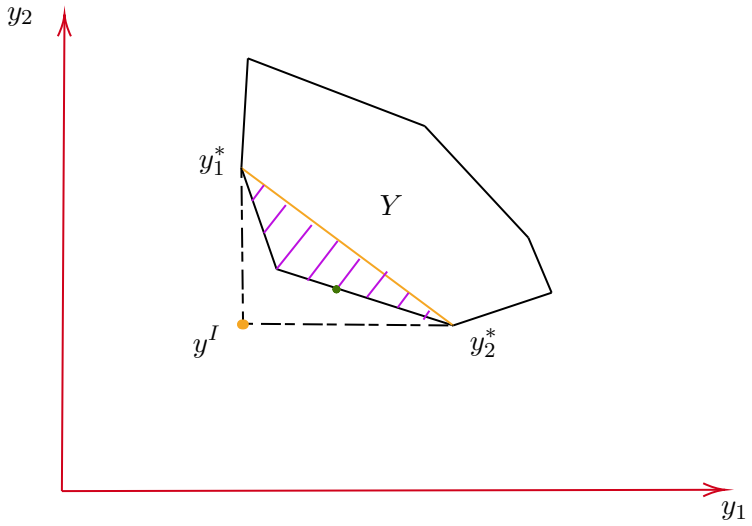

Figure 2.5: The CHIM is in this case the blue line connecting y_1^* and y_2^*. We sample points on this line and then send rays "downwards" to intersect with Y.

2.3 Revised NBI Method (RNBI)

The RNBI Method has been introduced by Shao and Ehrgott in [SE07]. It begins by constructing the Simplex S from Theorem 2.4 and uniformly sampling points on the subsimplex $\hat{S}$ (see Definition 2.5), just as the Global Shooting Method. In the next step, RNBI takes inspiration from NBI, and sends, starting from the sampled points on $\hat{S}$, rays which are perpendicular to $\hat{S}$ (i.e. parallel to the vector of ones: e) towards Y. Figure 2.6 visualizes the algorithm.

To compute the (first) intersection of the ray sent from the sampled point $q \in \hat{S}$ with Y, one can solve the following LP:

$$\min\{t : t \in \mathbb{R}_{\geq 0}, q + te \in Y\} \tag{2.8}$$

As Figure 2.6 shows, some of the rays may not intersect the boundary of Y at all. Consequently, the LP (2.8) may be infeasible. If feasible, the intersection point is given by $q + t^* e$, where t^* is the solution of the LP. Once again, we might encounter the issue that some of the points we receive are dominated. The following Theorem provides us with a simple algorithm to check nondominance:

Theorem 2.7 ([Ehr05])
Let $\overline{y} \in Y$ be a feasible point in objective space. Then $\overline{y}$ is nondominated, i.e. $\overline{y} \in Y_N$, if and only if $\overline{y}$ is optimal for the LP:

$$\min\{\langle e, y\rangle : y \leq \overline{y}, y \in Y\} \tag{2.9}$$

If $\overline{y}$ is dominated, then the LP provides us a nondominated solution dominating $\overline{y}$.

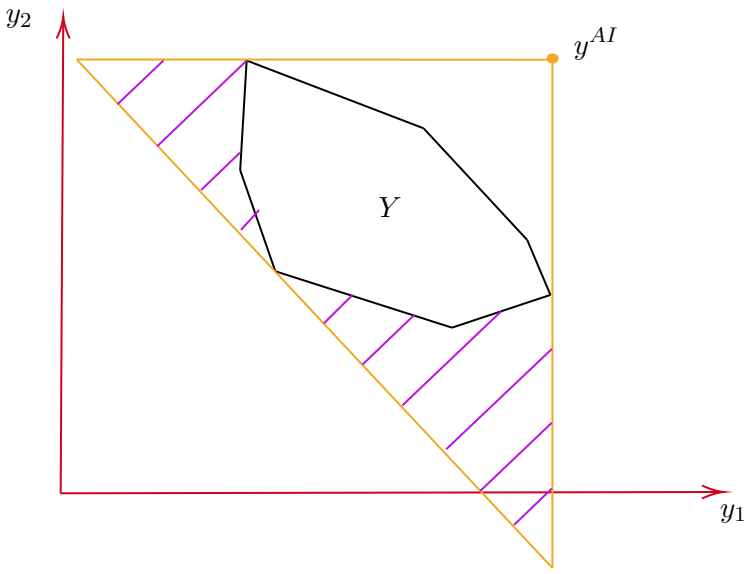

Figure 2.6: We send perpendicular rays from the sampled points on $\hat{S}$ towards Y. Some of these rays hit the boundary of Y.

To check nondominance for $\overline{y}$, we solve the LP (2.9) and obtain a solution y^*. One needs to be careful because of the following detail: We could have $y^* \neq \overline{y}$, yet $\langle e, y^*\rangle = \langle e, \overline{y}\rangle$, i.e. y^* and $\overline{y}$ are *distinct* optimal solutions of the LP. We therefore need to check whether $\langle e, y^*\rangle = \langle e, \overline{y}\rangle$ holds. If so, then $\overline{y}$ is nondominated; if not, then y^* is a nondominated solution dominating $\overline{y}$.

As anticipated, the RNBI Method has the advantage of having a provably low representation error and high uniformity, which implies that the discrete representation

R of Y_N it outputs is suitable for a human Decision Maker (DM). For technical details regarding these aspects we point out to Theorem 8 in [SE07].

We note that implementing RNBI requires solving four types of (single-objective) LPs. Assuming that we have an LP solver at our disposal, the rest of the implementation is quite simple. The LPs we need to solve are the following:

- LPs to compute the anti-ideal point y_{AI}, see (2.4).
- An LP to compute β, which is required for constructing the simplex. See Theorem 2.4.
- LPs to compute the intersection points, see (2.8).
- LPs for the nondominance check, see (2.9)

Chapter 3

Implementation

In this chapter we describe our implementation of RNBI for a concrete *Manpowering problem*. The scenario of this problem is, roughly, the following: A company needs to hire employees over a long period of time. The employees are structured hierarchically, and employees are periodically allowed to promote to a higher position (until they eventually reach a maximum). When someone gets hired, they usually lie at the bottom of the hierarchy, and start promoting from there. People may also leave the company at any time, independently of their position. The company needs to ensure that it has enough employees on each position at each time, but it is also constrained in the number of people it can employ, for example, by the yearly budget.

This problem has been already modeled as a multi-commodity min-cost-flow instance, which then can be written as a MOLP problem in the form introduced in Chapter 2. The precise definition of the Manpowering problem is not essential for us, because our implementation relies on a Black Box that abstracts from the problem and offers a suitable interface. Only the Black Box knows the concrete MOLP, and we do not have access to the solution space X, i.e. we do not know the precise variables the problem uses. However, the Black Box can solve the single-objective LPs required by the RNBI method (see end of Chapter 2) and we can therefore use it to implement the algorithm. This is possible because RNBI only requires access to the objective space.

In the next sections, we discuss the three main ingredients of our implementation. In Section 3.1 we explain how the Black Box can be programmed to solve the single-objective LPs we require for RNBI. In Section 3.2 we discuss our concrete RNBI implementation and also an extension which allows the user to have more flexibility in choosing which reference points to sample (on the subsimplex). In Section 3.3 we go into the visualization of the solutions produced by RNBI and explain how this can be useful for a Decision Maker (DM).

3.1 Programming the Black Box

Our RNBI implementation contains an abstract interface `Solver.java`, which specifies the functions we assume are given by any suitable Black Box:

- `int returnNumberOfObjectives()` returns the number of objectives of our MOLP (which we call p).
- `double[] getAntiIdealPoint()` computes the anti-ideal point and returns it as a vector (of size p) of doubles.
- `double getBeta()` returns β, as defined in Theorem 2.4.
- `Optional<Double> solveForReferencePoint(double[] refPoint)` calculates the (first) intersection point between the ray sent from the point `refPoint` and Y. This is implicitly done by solving the LP (2.8). The return type is `Optional<Double>` because this LP might be infeasible.
- `Optional<double[]> checkNondominance(double[] point)` checks whether the intersection point `point` is nondominated by solving the LP (2.9). The return type is `Optional<double[]>` to allow to return either nothing (if the point is nondominated) or a dominating point if `point` is dominated.

Defining this interface allows to implement RNBI for any problem for which we have a Black Box capable of implementing these functions. In this section we will concentrate on our concrete Black Box, provided to us for our concrete Manpowering problem. In our code, we have a class `SolverBlackBox.java` which inherits the interface `Solver.java` and implements all required functions using the Black Box.

To interact with the Black Box we use an Excel file named `Referenz.xlsx`. Modifying this file in various ways allows to solve the four types of single-objective LPs we need to solve. These are, as a reminder from Section 2.3:

- LPs to compute the anti-ideal point y^{AI}, i.e. maximising a fixed objective function.
- An LP to compute β, which is required for constructing the simplex, i.e. minimising the sum of the objectives.
- LPs to compute the intersection points, see (2.8).
- LPs for the nondominance check, see (2.9).

The aforementioned Excel file contains entries for all objective functions in different sheets. For the simplicity of the presentation, we will only show one of these sheets and pretend that it contains all the objective functions. Usually, this sheet looks like in Figure 3.1. The column *Zielfunktion* indicates the objective to which the entry corresponds. Hence, the first two entries correspond to objective 1 and the last two entries correspond to objective 2. In general, the column *Priorisierung* contains the weights (priorities) of each entry in their corresponding objective functions. In our

	A	B	C	D	E
1	Laufbahngruppe	Laufbahn	Steuerung	Zielfunktion	Priorisierung
2	UmP	FachD	Regeneration	1	1
3	UoP	FachD	Regeneration	1	1
4	UmP	FachD	Laufbahnwechsel	2	1
5	UoP	FachD	Laufbahnwechsel	2	1

Figure 3.1: Excel sheet describing objective functions. Standard form.

case, because all weights are set to 1, the first objective is given by the sum of the first two entries, while the second objective is given by the sum of the last two entries.

If we want to maximize *a specific* objective (required for computing y^{AI}), there are two things to do: first, we need to use negative weights, because the Black Box defaults to minimizing; second, we need to ignore the other objectives by setting the weights of their entries to 0. This is shown in Figure 3.2 for the objective function 1. If we want to minimize the sum of all objectives (required for computing β), it suffices to set all *Zielfunktion* entries to 1, without changing the weights. The Black Box will then consider all entries part of the same objective function and therefore minimize the sum of the real objective functions. This is shown in Figure 3.3.

	A	B	C	D	E
1	Laufbahngruppe	Laufbahn	Steuerung	Zielfunktion	Priorisierung
2	UmP	FachD	Regeneration	1	-1
3	UoP	FachD	Regeneration	1	-1
4	UmP	FachD	Laufbahnwechsel	2	0
5	UoP	FachD	Laufbahnwechsel	2	0

Figure 3.2: Maximizing objective function 1.

	A	B	C	D	E
1	Laufbahngruppe	Laufbahn	Steuerung	Zielfunktion	Priorisierung
2	UmP	FachD	Regeneration	1	1
3	UoP	FachD	Regeneration	1	1
4	UmP	FachD	Laufbahnwechsel	1	1
5	UoP	FachD	Laufbahnwechsel	1	1

Figure 3.3: Minimizing sum of objectives.

If we want to compute the (first) intersection point obtained by sending a ray from the reference point $q \in \mathbb{R}^p$ towards the objective space Y, we do not need to change anything from the standard configuration given in 3.1. We only need, on a special, different sheet, to provide the reference point q and the Black Box will automatically know what it needs to do. Performing the nondominance check is almost identical: we

only need to use another dedicated sheet to provide the point $\overline{y}$ whose nondominance we need to check.

Of course, for an automatic implementation of RNBI, we need to program the previously described behavior and to make it general. That is, we need to be able to modify the Excel file independently of the number of objectives p and of the configuration of the entries (i.e. how many entries belong to which objective, in what order are the entries appearing etc.) To do this, we used the programming language Java, and the library Apache POI API, which allows interacting with the Excel in the required ways. Our interface `Excel.java` specifies the required functions:

- `setUpMaximizeObjective(int objNo)` sets up the maximization of the objective function `objNo`. Again, Figure 3.2 is an example.
- `setUpCalculationOfBeta()` sets up the minimization of the sum of objectives. Again, Figure 3.3 is an example.
- `setUpReferenz(double[] referencePoint)` sets up the computation of the intersection point by reverting the Excel file to the standard configuration (see Figure 3.1) and writing the point `referencePoint` to sample from in its dedicated sheet.
- `setUpDominanzTest(double[] testPoint)` sets up the nondominance check by reverting the Excel file to the standard configuration (see Figure 3.1) and writing the point `testPoint` to check in its dedicated sheet.

After each such preparation, our implementation runs the Black Box, which reads `Referenz.xlsx` and solves the corresponding LP. We can then read the solution from another Excel file. The LPs computing the intersection points might be, as discussed in Chapter 2, infeasible. In this case the Black Box provides no output and we can also detect that.

3.2 Implementing RNBI

Now that we can solve the single-objective LPs, the implementation of RNBI is, with two exceptions, trivial. The algorithm works as follows: we compute y^{AI}, we compute β and the coordinates of the simplex, we sample points from the subsimplex, compute for each of those the intersection point and then check the resulting point (if it exists) for nondominance.

There is an issue specific to our Manpowering problem that affects the first step. Unfortunately, the anti-ideal point y^{AI} is not well defined for the problem, because our feasible set in objective space Y is, in fact, not a polytope, but only an unbounded

polyhedron. More precisely, one of the objective functions (called `penalty`) of our problem can grow arbitrarily large and the theory from Chapter 2 does not fully apply. To fix this issue, we minimize this objective function first, obtaining some value $v_{\min}$, and then use for the corresponding entry in the anti-ideal point the value $C \cdot v_{\min}$, where C is some user-specified constant. Geometrically, this corresponds to adding an (artificial) new constraint, which cuts the infinitely large polyhedron and turns it into a (bounded) polytope, thus allowing us to use RNBI as discussed in Section 2.3. This works out quite well, because in practice, values beyond a certain threshold for this objective function are not interesting. Therefore, the Decision Maker (DM) can choose a suitable C (which we call `penaltyFactor`) and not worry about this aspect too much.

The only other non-trivial step in the algorithm is the sampling. In general, we are given a subsimplex of dimension $p-1 \geq 1$, defined by p points $v^1, \ldots, v^p$. We recall Definition 2.5: sampling uniformly on $\hat{S}$ with parameter $m \geq 1$ means constructing the set:

$$\mathrm{Unif}(\hat{S}, m) := \left\{ \sum_{k=1}^{p} \alpha_k v^k : \alpha_k \text{ multiple of } \eta, \ \sum_{k=1}^{p} \alpha_k = 1 \right\} \tag{3.1}$$

where $\eta := 1/m$. To do this, we note that it suffices to iterate over the set:

$$\mathrm{Part}(k, m) := \left\{ (a_1, \ldots, a_k) \in \mathbb{Z}^k_{\geq 0} : \sum_i a_i = m \right\} \tag{3.2}$$

This is because, for any tuple $(a_1, \ldots, a_k) \in \mathrm{Part}(k, m)$, we can then take $(\alpha_1, \ldots, \alpha_k)$ $\in \mathrm{Unif}(\hat{S}, m)$ given by $\alpha_k := a_k/m$. This procedure provides an iterator over all tuples $(\alpha_1, \ldots, \alpha_k)$ that are needed to construct $\mathrm{Unif}(\hat{S}, m)$. Implementing an iterator for $\mathrm{Part}(k, m)$ is not too difficult. We show our implementation in the Appendix A.

When using RNBI for our concrete problem, we noticed that relatively few sampled points lead to a feasible solution, i.e. the LP (2.8) is often infeasible. There exist clear areas on the subsimplex where no sampled point leads to a solution. The user might notice this and want to avoid those areas for future runs of the algorithm. We therefore implemented an extension of RNBI, which allows the user to specify an input in the following form:

$$\text{Input} := \left((P^i, m_i, d_i) : P^i \in \mathbb{R}^p_{\geq 0}, \ \sum_{j=1}^{p} P^i_j = 1, \ m_i, d_i \in \mathbb{Z}_{\geq 1} \right)_{i=1}^{N} \tag{3.3}$$

Hereby the i-th entry of the Input is given by a tuple (P^i, m_i, d_i). P^i determines the sampled point $q_i \in \hat{S}$. This point is the convex combination given by $q_i := \sum_k P^i_k v^k$.

m_i and d_i control, respectively, how close to q_i and how many points around q_i we sample. Let $\eta_i := 1/m_i$. To avoid notation clutter we fix i and use P for P^i, q for q_i and η for η_i. We call a vector $\gamma \in \mathbb{Z}^p$ *d-deep* iff $\sum_j \gamma_j = 0$ and $\sum_{j:\gamma_j>0} \gamma_j \leq d$. The following set contains the coefficients we will use to sample points around q:

$$\text{Coefs}(P, m, d) := \{P' := (P_k + \gamma_k \eta)_{k=1}^p : \ 0 \leq P' \leq 1, \ \gamma \in \mathbb{Z}^p \text{ is } d\text{−}deep\} \tag{3.4}$$

We go by units of $\eta = 1/m$ in all directions. Because the values in γ sum up to 0, each vector $P' \in \text{Coefs}(P, m, d)$ has (non-negative) values which sum up to 1 and therefore can be used to form a convex combination. The fact that γ is *d-deep* controls how deep we can go around q, while m controls how small the steps are in which we sample around q. The points we sample are then naturally given by:

$$\text{Unif}(P, m, d) := \left\{ \sum_k P'_k v^k : P' \in \text{Coefs}(P, m, d) \right\} \tag{3.5}$$

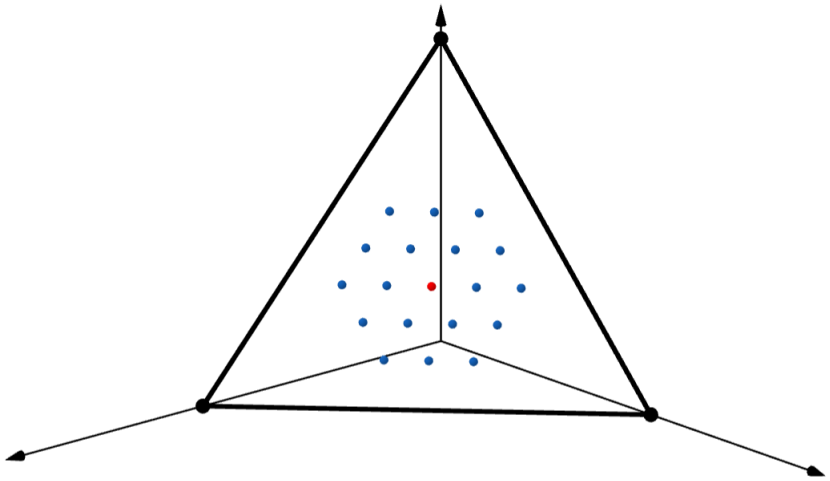

Figure 3.4: Example of custom sampling around the red point q for a special case where $p = 3$, subsimplex given by corners $\{(0, 0, 1), (0, 1, 0), (0, 0, 1)\}$, $q := (1/3, 1/3, 1/3)$, $m := 10$ and $d := 2$.

An example is shown in Figure 3.4, where we assumed for simplicity that the subsimplex has precisely the corners $\{(0, 0, 1), (0, 1, 0), (0, 0, 1)\}$ ($p = 3$), such that the sets Coefs and Unif coincide. The red point is $q := (1/3, 1/3, 1/3)$, and we used $m := 10$ and $d := 2$. We then sample both the red point and the blue points.

In practice, we have to compute the sets $\text{Coefs}(P^i, m_i, d_i)$ for all entries in the input, and then compute the union of these sets while ensuring that we eliminate duplicates.

Our implementation does precisely this and then offers an iterator over all the resulting coefficient vectors, which can then be used ad-hoc to compute the corresponding sampled points. To compute the disjoint union of the sets Coefs we point out to the Appendix A, where pseudo-Java code for this task is provided. To generate the *d-deep* γ vectors we rely on the previously designed iterator for Part$(\cdot, \cdot)$.

Our implementation also supports running both RNBI and the newly introduced custom sampling at the same time. This means that we sample both the points determined by the RNBI and the points selected by the user. The user can determine the precise behavior by using the `configRNBI.json` file. This is also where the `penaltyFactor` entry required for the computation of y^{AI} is provided and the parameter m which determines how many points we sample for standard RNBI.

Moreover, we have also implemented RNBI for a smaller, Demo problem, which is independent from the "real" Manpowering problem. The Demo functionality might be interesting for users who want to develop intuition regarding the RNBI method. The Demo problem (Example 6.3 from [Ehr05]) is simple:

$$\begin{aligned} \min \quad & 3x_1 + x_2 \\ \min \quad & -x_1 - 2x_2 \\ & x_2 \leq 3 \\ & 3x_1 - x_2 \leq 6 \\ & x_1, x_2 \geq 0 \end{aligned} \tag{3.6}$$

Ehrgott also provides the solution in [Ehr05], and the user can check that our program works correctly. This reality check would be much harder with the bigger Manpowering problem, which provides another good reason to include the Demo problem. Solving the single-objective LPs is in this case no longer done with the Black Box (which only works for the Manpowering problem), but with the well-known library Gurobi [Gur22].

3.3 Visualizing the results

The RNBI implementation computes, for each sampled point $q \in \mathbb{R}^p$, the corresponding intersection with the polytope Y and, if it exists, also checks the nondominance of the resulting point q_{hit}. If the point is dominated, it also outputs the point which dominates it. All these results are gathered in a `.csv` file which is then inputted to a Python program. Our Python program uses the well known library Dash/Plotly [Plo15] for visualizing the results. The examples shown in this section can be consulted at `https://github.com/RaduVintan/IDP`.

The purpose of our visualization is twofold: we want both to understand RNBI by inspecting its output for different configurations, and to offer a suitable interface

(+ enough information) for a Decision Maker (DM) looking to choose a solution. To make the visualization of the objective space relatively intuitive we will always assume $p \leq 3$. We start with a relatively small example: a solution obtained for the Demo problem at (3.6). The output appears in Figure 3.5. There are 11 reference points and, as expected, 2 objectives. The three black reference points on the subsimplex (i.e. the line opposite to the anti-ideal point) indicate infeasibility, i.e. their corresponding intersection LP has no solution. The other reference points all produced solutions. We connect the reference points to their corresponding *hit* point and this why we can see the blue lines. There exists a menu (not shown in the Figure) where the user can control what the figure shows. We can deactivate or activate the showing of reference points, (dominated or nondominated) hit points, lines connecting reference points to hit points etc. In this particular case, all points we found passed the nondominance check. We will see what happens if this is not the case in the next example.

We now show a solution for a variant of the Manpowering problem with 3 objectives. The output appears in Figure 3.6. For clarity, we only show the hit points and not the reference points. To receive details about any point, a hovering functionality is available, as shown in Figure 3.7, where we see the 3 coordinates of a certain hitpoint, corresponding to the 3 objective functions: *Penalty*, *Regeneration*, and *Laufbahnwechsel*. The hit point has failed the nondominance check and we are informed that it is dominated. To see which point dominates a hit point, we can use the menu and activate an option that connects all hit points to their corresponding dominating points (as found by the nondominance check). Figure 3.8 shows an "ideal" output, consisting only of nondominated solutions. The blue points are hit points which have passed the nondominance check. The other hit points have failed the check, and we do not show them. However, as noticed in Theorem 2.7, each failed nondominance check for a hit point $\overline{y}$ outputs a nondominated solution $y^* \neq \overline{y}$ dominating $\overline{y}$ and certifying that $\overline{y}$ is dominated. The red points in the Figure are such y^* solutions, found during the (failed) nondominance checks.

The last type of visualization we implemented is given by Figure 3.9. Here, the objective function *Penalty* is (only) implicitly shown, using colors. Bright red indicates high penalty, bright green indicates low penalty. The two remaining axes correspond to the other two objective functions: *Regeneration* and *Laufbahnwechsel*. We again show the ideal output, consisting of only the nondominated solutions. We can no longer use colors (blue and red) to differentiate between the two types of nondominated solutions as we did in the previous example, but the menu can be used to show or hide the type(s) of points we are interested in. The menu also permits showing or hiding the projected simplex. We suppress showing it in this case to make the Figure smaller. The projected visualization is often easier on the eyes than the 3D one, especially if there are many solutions to choose from.

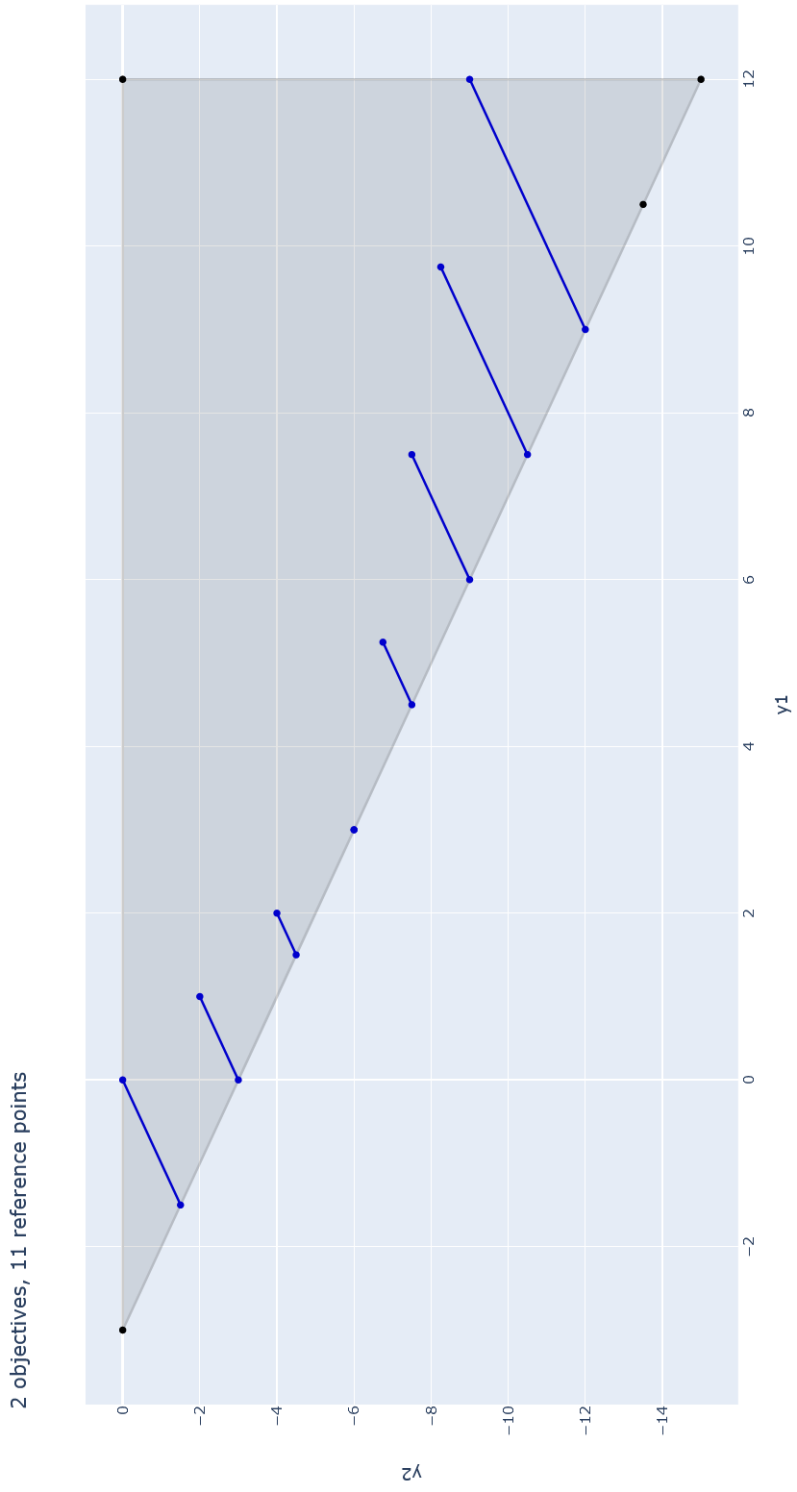

Figure 3.5: 2D Example for the Demo problem

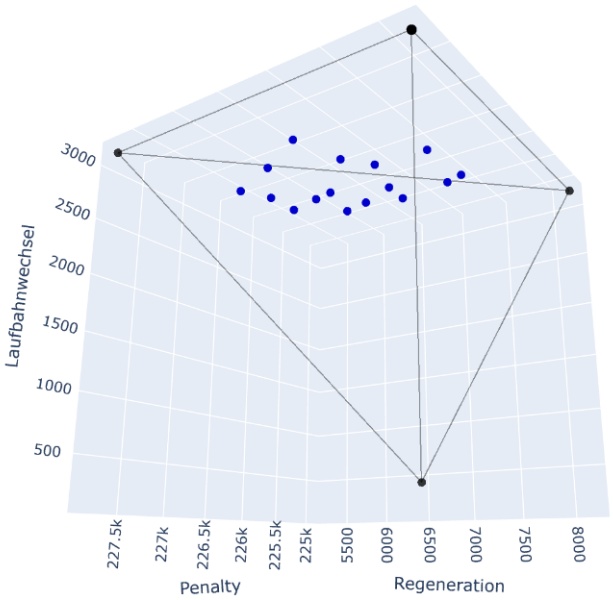

Figure 3.6: 3D Example for the Manpowering problem

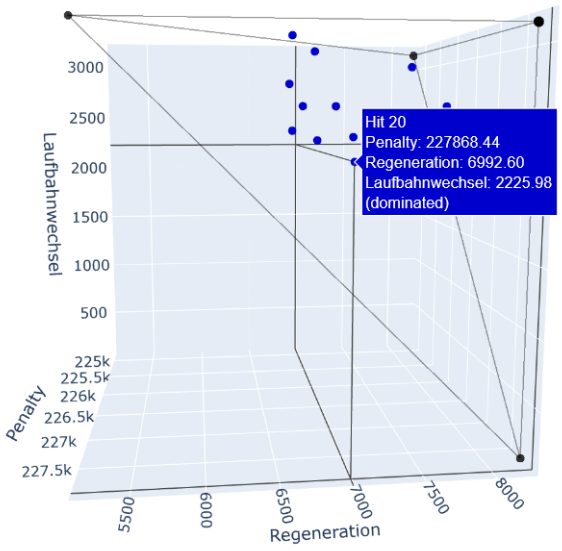

Figure 3.7: Hovering over reference or hit points shows several details

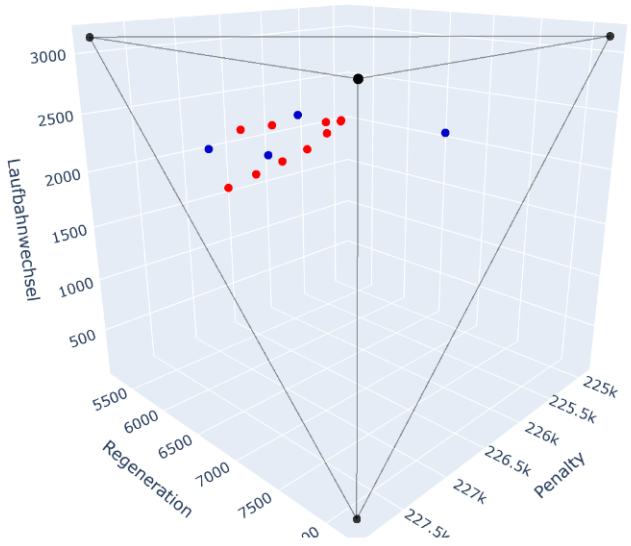

Figure 3.8: 3D Example showing only the nondominated solutions

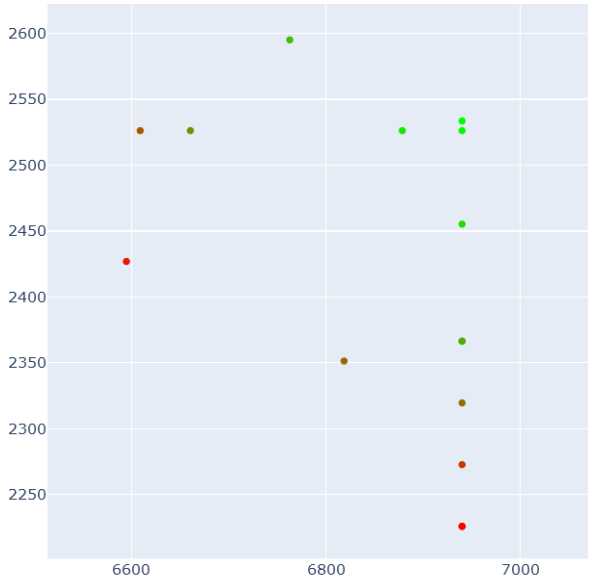

Figure 3.9: Projecting the 3D Example into 2D and representing the *Penalty* objective function using colors. The lonely blue hit point from the previous Figure is not shown to keep this Figure small.

Chapter 4

Conclusion

We introduced three well known algorithms to determine a suitable discrete representation R of the set Y_N of nondominated solution of an MOLP. These algorithms are the *Global Shooting* method (Section 2.1), the *Normal Boundary Intersection (NBI)* method (Section 2.2) and the *Revised NBI (RNBI)* method (Section 2.3). Only the latter algorithm allows us to tune both the *representation error* and the *uniformity* of R. We implemented RNBI both for a small Demo MOLP and for a middle-sized *Manpowering problem* given to us as a Black Box (Section 3.1). We also extended the RNBI implementation by allowing the user to use custom sampling (Section 3.2). Our visualization, implemented in Plotly/Dash, allows a Decision Maker (DM) to inspect different properties of the obtained representation R and to easily decide on a solution (Section 3.3).

Our implementation, while practically useful and interesting, has several limitations, which we believe might be worth addressing in a future project. Most notably, it would be very convenient to implement an interactive RNBI solver, where the Decision Maker (DM) can check the visualization on the fly, while it is being constructed, and spontaneously decide where to sample future points, how many new reference points to sample in which area etc. Our implementation is static and the user has to wait for all computations to end before being able to see the visualization and then to run the program again with different parameters. Waiting for a run to end takes quite a while for larger problems and is not efficient. Moreover, user friendliness could be increased if the DM is able to give online input on the visualization itself, without needing to use a cumbersome `.json` file or other text format.

Appendix A

Remarks on Implementation

The Java code below iterates over the set $\mathrm{Part}(k, m)$ given in (3.2), assuming that we start from $a := [m, 0, \ldots, 0]$. Note that a is 0-indexed.

```
static int[] next(int[] a, int m) {
  int k = a.length;
  if(a[k - 1] == m) {
    return null;
  }

  int index = k - 1;
  while(a[index] == 0) {
    index--;
  }

  if(index == k - 1) {
    int aux = a[index];
    a[index] = 0;

    int otherindex = index - 1;
    while(a[otherindex] == 0) {
      otherindex--;
    }
    a[otherindex]--;
    a[otherindex + 1] = 1 + aux;
    return a;
  }

  a[index]--;
  a[index + 1] = 1;
  return a;
}
```

The following pseudo-Java code generates the disjoint union of the sets Coefs (see (3.4)). We assume that the function `addToSamples` takes care of eliminating duplicates. We use the previous iterator. Again, all vectors are 0-indexed.

```
void generateCoefs() {
  for(int i = 0; i < inputSize; i++) {
    // P := P_i, given by user
    // eta := 1/m_i, where m_i given by user
    // d := d_i, given by user

    double[] P_prime = new double[p];

    /* r := How many multiples of +eta (and -eta) do we add to
     * our coefficients.
     */
    for(int r = 1; r <= d; r++) {
     /* gamma is a vector of size p with non-negative integer entries,
      * s.t. the sum of these entries is r.
      * Corresponds to adding multiples of eta.
      *
      * Later in the code we also add negative entries
      * to gamma to make it d-deep.
      */
      int[] gamma = new int[p];
      gamma[0] = r;
      while(true) {
        /* The idea is now to add the r many multiples of -eta. For
         * this we need to use the positions where gamma = 0.
         */
        ArrayList<Integer> zeroIndices = new ArrayList<>();

        for(int j = 0; j < p; j++) {
          if(gamma[j] == 0) {
            zeroIndices.add(j);
          }
        }

        int howManyZero = zeroIndices.size();
        // We can only do this if gamma has some zero entries
        if(howManyZero != 0) {
          /* aux is a vector of size howManyZero with non-negative
```

```
          * integer entries, s.t. the sum of these entries is r.
          * Corresponds to adding multiples of -eta.
          */
          int[] aux = new int[howManyZero];
          aux[0] = r;
          while(true) {
             // Modify gamma using aux to make it d-deep.
             for(int j = 0; j < howManyZero; j++) {
               gamma[zeroIndices.get(j)] = -aux[j];
             }
             // Generate the corresponding coefficient vector.
             for(int j = 0; j < p; j++) {
               P_prime[j] = P[j] + gamma[j] * eta;
             }
             // Return to initial gamma.
             for(int j = 0; j < howManyZero; j++) {
               gamma[zeroIndices.get(j)] = 0;
             }
             // Add new coefficient vector.
             addToSamples(P_prime);

             /* Generate new aux vector with non-negative entries
              * summing to r using previous iterator.
              * We break if no next vector exists.
              */
             next(aux, r);
           }
         }
         /* Generate new gamma vector with non-negative entries
          * summing to r using previous iterator.
          * We break if no next vector exists.
          */
         next(gamma, r);
       }
     }
   }
}
```

List of Figures

Bibliography

[BS97] H. Benson and S. Sayin. *Towards Finding Global Representations of the Efficient Set in Multiple Objective Mathematical Programming*. New York, NY, 1997.

[DD98] I. Das and J. E. Dennis. "Normal-Boundary Intersection: A New Method for Generating the Pareto Surface in Nonlinear Multicriteria Optimization Problems". In: *SIAM J. on Optimization* 8.3 (1998).

[Ehr05] M. Ehrgott. *Multicriteria Optimization*. Springer, 2005.

[Gur22] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2022. URL: https://www.gurobi.com.

[LER17] K.-M. Lin, M. Ehrgott, and A. Raith. "Integrating column generation in a method to compute a discrete representation of the non-dominated set of multi-objective linear programmes". In: *A Quarterly Journal of Operations Research (4OR)* 15 (2017), pp. 331–357.

[Plo15] Plotly Technologies Inc. *Collaborative Data Science*. 2015. URL: https://plot.ly.

[Roc70] T. Rockafellar. *Convex analysis*. Princeton Mathematical Series. Princeton, N. J.: Princeton University Press, 1970.

[Rua20] Y. Ruan. "Generating a Representative Solution Set for Multi-Objective Optimization Problems: An Application of the RNBI Method to A Multi-Commodity Flow Problem". In: *Master's Thesis. Technische Universität München* (2020).

[SE07] L. Shao and M. Ehrgott. "Finding Representative Nondominated Points in Multiobjective Linear Programming". In: *2007 IEEE Symposium on Computational Intelligence in Multi-Criteria Decision-Making*. May 2007, pp. 245–252.