MIE335 Project (Winter 2023):

Multiobjective Multidimensional 0-1 Knapsack Problem

Contents

Key concepts: Algorithm implementation, computational considerations, algorithm performance analysis, multiobjective integer programming, knapsack problem

1 Multidimensional 0-1 Knapsack Problem

Given a set of items ($i = 1, ..., n$), each with a utility c_i, a weight w_i, the **0-1 knapsack problem** aims to determine which items to include in a collection so that the total weight can not exceed the capacity W and the total utility is maximized.

We can formulate the 0-1 knapsack problem (KP), as an integer program, as follows:

$$\text{(KP):} \quad \max \quad \sum_{i=1}^{n} c_i x_i \tag{1a}$$

$$\text{s.t} \quad \sum_{i=1}^{n} w_i x_i \leq W \tag{1b}$$

$$x_i \in \{0, 1\}, \ i = 1, ..., n \tag{1c}$$

where x_i is a binary decision variable taking the value of 1 if item i is selected in the collection and 0 otherwise. The constraint (1b) is referred to as a *knapsack constraint*.

The **multidimensional 0-1 knapsack problem** (MKP) generalizes the 0-1 knapsack problem by considering multiple knapsack constraints. (Note that dimension does not refer to the shape of any items.) More specifically, an m-dimensional 0-1 knapsack problem (m-KP) has m knapsack constraints, and can be formulated as an integer program as follows:

$$(m\text{-KP}): \quad \max \quad \sum_{i=1}^{n} c_i x_i \tag{2a}$$

$$\text{s.t} \quad \sum_{i=1}^{n} a_{ik} x_i \leq b_k, \quad k = 1, \ldots, m \tag{2b}$$

$$x_i \in \{0, 1\}, \qquad i = 1, \ldots, n \tag{2c}$$

where x_i is a binary decision variable taking the value of 1 if item i is selected in the collection and 0 otherwise as before, b_k is the capacity of knapsack k, and a_{ik} denotes the resource consumption amount of item i in the k^{th} knapsack. You can think of a case where there are m different resources with capacities $b_1, \ldots, b_m$, and the goal is to find a subset of items that yields maximum profit without exceeding the resource capacities.

The MKP is one of the most well-known constrained integer programming problems, and the large domain of its applications has greatly contributed to its fame. For instance, MKP modeling has been used in capital budgeting, project selection, cutting stock, loading problems, investment policy development, allocation of databases and processors in a distributed data processing, delivery of groceries in vehicles with multiple compartments, approval voting, and daily management of a remote sensing satellite [1].

2 Multiobjective Optimization

A **multiobjective optimization problem** with feasible set $\mathcal{X} \subseteq \mathbb{R}^n$ and $J \in \mathbb{Z}_+, J \geq 2$ objective functions (or criteria) $g_j : \mathcal{X} \to \mathbb{R}, j = 1, \ldots, J$ can be written as

$$\text{(MOP)}: \min_{x \in \mathcal{X}} \; g(x) := \{g_1(x), \ldots, g_J(x)\} \tag{3}$$

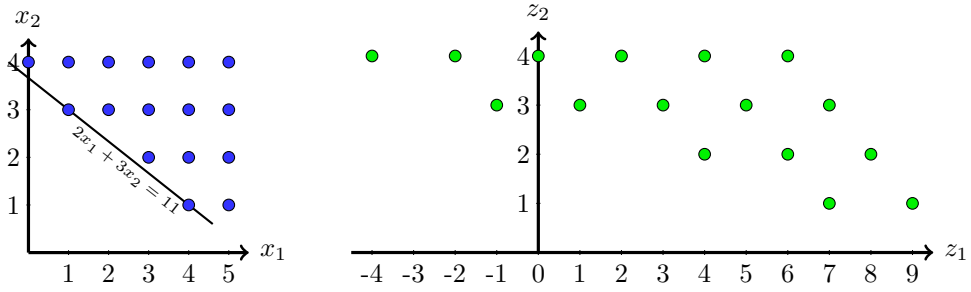
where $g = (g_1, \ldots, g_J) : \mathcal{X} \to \mathbb{R}^J$ is the *objective function vector*, which maps the feasible set defined in the *decision space* to the *objective space*. The codomain of g, i.e., $\mathbb{R}^J$, is called the *criterion space*. The image of the feasible set is denoted by $\mathcal{Z} := g(\mathcal{X}) := \{z \in \mathbb{R}^J : \exists\, x \in \mathcal{X} \text{ s.t. } z = g(x)\}$ and usually referred to as the *feasible set in criterion space*.

Consider the following bi-objective integer linear programming example:

$$\begin{aligned} \text{(MOP-Ex)}: \quad \min \; & \{2x_1 - x_2, x_2\} \\ \text{s.t. } & 2x_1 + 3x_2 \geq 11 \\ & x_1 \leq 5 \\ & x_2 \leq 4 \\ & x_1, x_2 \in \mathbb{Z}_+ \end{aligned}$$

That is, we have:

- $\mathcal{X} = \{x \in \mathbb{Z}^2_+ : 2x_1 + 3x_2 \geq 11, x_1 \leq 5, x_2 \leq 4\} \to$ feasible set in decision space
- $g_1(x) = 2x_1 - x_2 \to$ first objective function
  $g_2(x) = x_2 \to$ second objective function
- $g(x) = (2x_1 - x_2, x_2) \to$ objective function vector
- $\mathcal{Z} = \{z \in \mathbb{R}^2 : \exists\, x \in \mathcal{X} \text{ s.t. } z_1 = 2x_1 - x, z_2 = x_2\} \to$ feasible set in criterion space

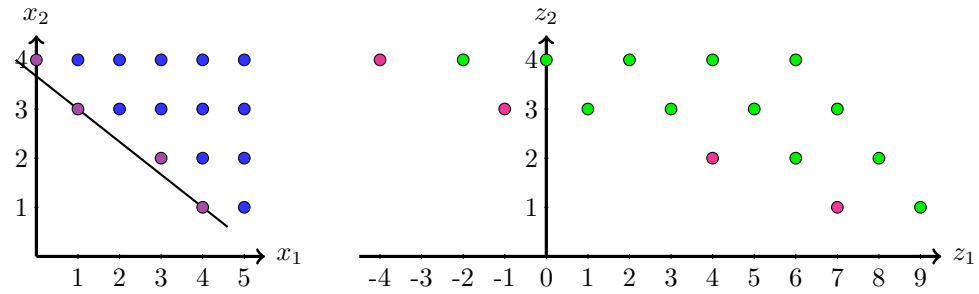


(a) $\mathcal{X}$ of (MOP-Ex) (b) $\mathcal{Z}$ of (MOP-Ex)

Figure 1: Feasible sets in decision and criterion spaces of (MOP-Ex)

In multiobjective optimization, the "optimal value" is a set, rather than a single point, often called the **nondominated frontier** or *Pareto frontier*. It is defined to be the set of vectors, $z \in \mathbb{R}^J$, in the criterion space, having the property that (i) z is the criterion-space image of some feasible solution, i.e., $z = g(x)$ for some $x \in \mathcal{X}$, or $z \in \mathcal{Z}$, in which case we say z is *feasible*, and (ii) there does not exist any other feasible solution, $z' \in \mathcal{Z}$, which *dominates* z, i.e., for which $z'_j \leq z_j$ for all $j = 1, \ldots, J$ and $z'_j < z_j$ for at least one index $j \in \{1, \ldots, J\}$. An element of the nondominated frontier (NDF) is known as a **nondominated point** (NDP). In other words, an NDP is a feasible objective vector for which none of its components can be improved without making at least one of its other components worse. The union of preimages of NDPs is called the *efficient set*, whose elements are the *efficient solutions*. So, a feasible solution $x \in \mathcal{X}$ is *efficient* (or Pareto optimal) if its image $z = g(x)$ is an NDP, i.e., *nondominated*.

Figure 2 shows the efficient set and NDF for our example problem (MOP-Ex). For instance, the point $\hat{z} = (7, 1)$, which is the image of $\hat{x} = (4, 1)$, i.e., $\hat{z} = g(\hat{x})$, is an NDP as (i) it is feasible, i.e., $\hat{z} \in \mathcal{Z}$, and (ii) there does not exist any $z' \in \mathcal{Z}$ that dominates it, i.e., $\{z' \in \mathcal{Z} : (z'_1 \leq \hat{z}_1 \text{ and } z'_2 < \hat{z}_2) \text{ or } (z'_1 < \hat{z}_1 \text{ and } z'_2 \leq \hat{z}_2)\} = \emptyset$. On the other hand, $z = (9, 1)$ and $z = (8, 2)$ are not on the NDF as they are dominated by $\hat{z} = (7, 1)$.



(a) Efficient set of (MOP-Ex) (b) NDF of (MOP-Ex)

Figure 2: Efficient set and nondominated frontier of (MOP-Ex)

Two useful constructs are the so-called *ideal point* and *supernal point*, denoted by z^I and z^S, whose components are obtained by minimizing and maximizing individual objective functions over the feasible set of the problem, so $z^I_j := \min\{z_j : z \in \mathcal{Z}\}$ and $z^S_j := \max\{z_j : z \in \mathcal{Z}\}$ for all $j = 1, \ldots, J$, respectively. As such, the NDF is contained in the **hyperrectangle** defined by z^I and z^S, i.e., NDF $\subseteq \{z \in \mathbb{R}^J : z^I_j \leq z_j \leq z^S_j, \forall j = 1, \ldots, J\}$. Figure 3 illustrates these concepts for our bi-objective example.

An NDP can be found in a variety of ways. One of the most commonly used is the **weighted-sum method**, which solves an optimization problem with a single objective obtained as a positive (convex) combination of the objective functions of the multiobjective

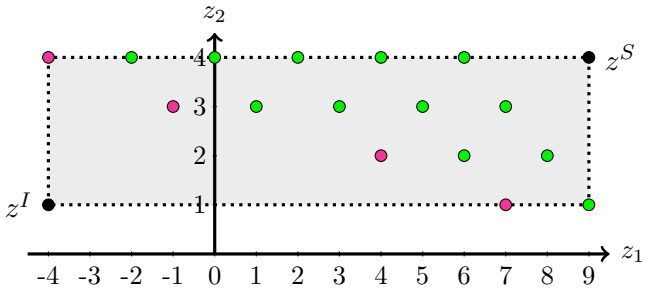


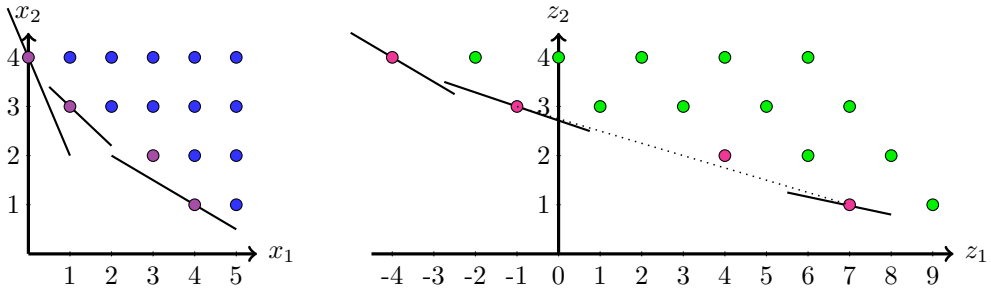
Figure 3: Ideal and supernal points of (MOP-Ex), and the rectangle defined by them

problem:

$$\min_{x \in \mathcal{X}} \sum_{j=1}^{J} \lambda_j g_j(x) \tag{5}$$

where $\lambda_j > 0$ for all $j = 1, \ldots, J$. For any *positive* weight vector λ, any optimal solution of (5) is an efficient solution for (MOP), given in (3), i.e., its image is nondominated. Such a solution and its image are called a *supported efficient solution* and a *supported nondominated point*, respectively. Thus, an efficient solution x is supported if there exists a positive vector λ for which x is an optimal solution of (5), otherwise x is called *unsupported*.

- $\lambda = \left(\frac{1}{6}, \frac{5}{6}\right) \Rightarrow \min\left\{\frac{2}{6}x_1 + \frac{4}{6}x_2 : x \in \mathcal{X}\right\} \Rightarrow x^* = (4,1) \Rightarrow z^* = (7,1) \rightarrow$ supported
- $\lambda = \left(\frac{2}{9}, \frac{7}{9}\right) \Rightarrow \min\left\{\frac{4}{9}x_1 + \frac{5}{9}x_2 : x \in \mathcal{X}\right\} \Rightarrow x^* = (1,3) \Rightarrow z^* = (-1,3) \rightarrow$ supported
- $\lambda = \left(\frac{1}{3}, \frac{2}{3}\right) \Rightarrow \min\left\{\frac{2}{3}x_1 + \frac{1}{3}x_2 : x \in \mathcal{X}\right\} \Rightarrow x^* = (0,4) \Rightarrow z^* = (-4,4) \rightarrow$ supported
- $x = (3,2)$ can **not** be obtained in this way $\Rightarrow z = (4,2) \rightarrow$ unsupported



(a) In decision space (b) In criterion space

Figure 4: Some single weighted objective optimizations for (MOP-Ex)

Another way to find an NDP is to optimize with respect to each objective function in turn, in an hierarchical manner. More specifically, we can solve **lexicographic optimization problems**, by minimizing one objective at a time, sequentially, and using optimal objective values of solved problems as constraints in the next ones. For example, for the order $1, \dots, J$, first determine $\hat{z}_1 := \min\{g_1(x) : x \in \mathcal{X}\}$, and then for each $j = 2, \dots, J$, in turn, sequentially solve

$$\hat{z}_j := \min\{g_j(x) : x \in \mathcal{X} \text{ and } g_{j'}(x) \leq \hat{z}_{j'}, j' = 1, \dots, j-1\}.$$

Then the vector $\hat{z} := (\hat{z}_1, \dots, \hat{z}_J)$ is an NDP. We will represent the above lexicographical optimization problem to find $\hat{z}$ as

$$\operatorname{lex\,min}_{x \in \mathcal{X}} (g_1(x), \dots, g_J(x)), \tag{6}$$

or rather as

$$\operatorname{lex\,min}_{z \in \mathcal{Z}} (z_1, \dots, z_J). \tag{7}$$

Note that parentheses in (6) and (7) signify that the objective functions are ordered, whereas curly brackets in (3) denote that the objective functions are given as an unordered set.

For our bi-objective example, as illustrated in Figure 5, we can discover the following NDPs via the two possible lexicographic optimizations:

- $\operatorname{lex\,min}_{x \in \mathcal{X}}(g_1(x), g_2(x)) \Rightarrow z^* = (-4, 4)$
- $\operatorname{lex\,min}_{x \in \mathcal{X}}(g_2(x), g_1(x)) \Rightarrow z^* = (7, 1)$

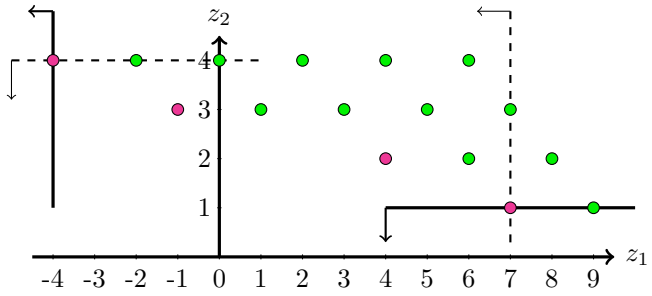


Figure 5: NDPs obtained via lexicographic optimizations for (MOP-Ex)

The goal of this project is to (efficiently) generate the NDF of the multidimensional 0-1 knapsack problem. You will implement three main algorithms that are outlined next.

3 Algorithms to Generate Nondominated Frontier

In this section, we will use a hypothetical bi-objective integer programming example whose feasible set in the criterion space is given in Figure 6. In the figure, the dotted points show the integer lattice, circle points represent the feasible points in the objective space, while the pink circle points illustrate the NDF of this example problem.

Throughout this section, we will make the convention that circle and square points will represent the points discovered an undiscovered by an algorithm at an iteration where the snapshot is taken, respectively. Moreover, the discovered and undiscovered NDPs will be highlighted in pink and blue, respectively.

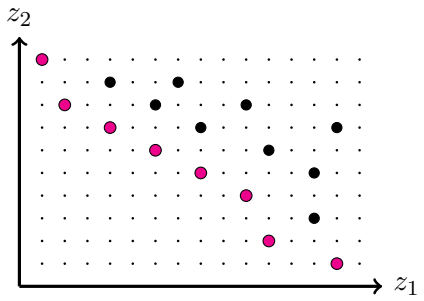


Figure 6: Criterion space of a hypothetical example

3.1 Brute-force Enumeration Method

This is the most naive algorithm we can design to find the NDF. The algorithm first enumerates **all** the feasible solutions in the decision space, then finds their objective images, removes copies (as a feasible point in the criterion space might have multiple pre-images), and finally computes the nondominated ones out of the remaining points.

Algorithm 1 Brute-force Enumeration

Input: A MOP as in (3)

Output: NDF as $\mathcal{Z}^N$

1: Enumerate all points in $\mathcal{X}$

2: Find feasible images, i.e., let $\mathcal{Z} = \bigcup_{x \in \mathcal{X}} \{g(x)\}$

3: $\mathcal{Z}$ = RemoveDuplicates($\mathcal{Z}$)

4: $\mathcal{Z}^N$ = RemoveDominated($\mathcal{Z}$)

In our running example of Figure 6, this method will have all 17 points enumerated at the end of Step 3, then eliminate the dominated ones to return the 8 points on the NDF.

3.2 Rectangle Division Method for Problems with Two Objectives

This method only applies to problems with two objectives. It conducts a search of NDPs in the criterion space by dividing it into smaller regions. The search regions take the form of a rectangle, where a rectangle is represented by two points, namely its northwest (NW) and southeast (SE) corner points and denoted by $R[\text{NW}, \text{SE}]$. The search of an NDP in a rectangle relies on a lexicographical optimization problem, see for instance (6) or (7). Note that for the ease of exposition we will use (7) rather than (6), however, when we are solving them, we will be indeed solving optimization problems associated with (6). In what follows, we will first explain the basics of this method on the hypothetical example provided in Figure 6, and then provide the full algorithm (Algorithm 2).

The algorithm starts by solving the two lexicographical optimization problems

$$\bullet \ \operatorname*{lex\,min}_{z \in \mathcal{Z}} (z_1, z_2) \to z^{NW} \qquad \bullet \ \operatorname*{lex\,min}_{z \in \mathcal{Z}} (z_2, z_1) \to z^{SE}$$

to define the initial rectangle

$$R_1 = R[z^{NW}, z^{SE}] = \{z \in \mathbb{R}^2 : z_1^{NW} \leq z_1 \leq z_1^{SE} \text{ and } z_2^{SE} \leq z_2 \leq z_2^{NW}\}$$

given in Figure 7a. Then, as shown in Figure 7b, the initial rectangle is divided into two rectangles R_2 and R_3, e.g., via bisecting along the z_2 axis:

$$\bullet \ R_2 = R[z^{MW}, z^{SE}] \qquad \bullet \ R_3 = R[z^{NW}, z^{ME}]$$

where the MidWest point z^{MW} and the MidEast point z^{ME} correspond to the endpoints of the bisecting line segment:

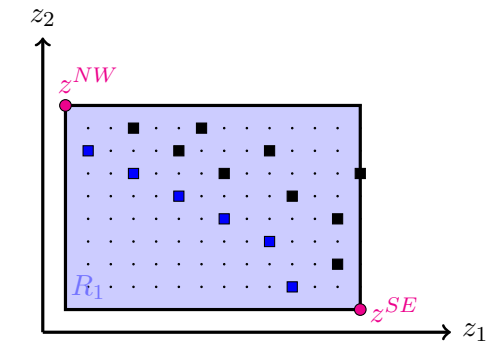
$$z^{MW} = \left(z_1^{NW}, \frac{z_2^{SE} + z_2^{NW}}{2}\right) \ , \ z^{ME} = \left(z_1^{SE}, \frac{z_2^{SE} + z_2^{NW}}{2}\right)$$

Next, the algorithm picks one of the two rectangles, say R_2, and searches for an NDP that belongs to that rectangle by solving another lexicographical optimization problem:

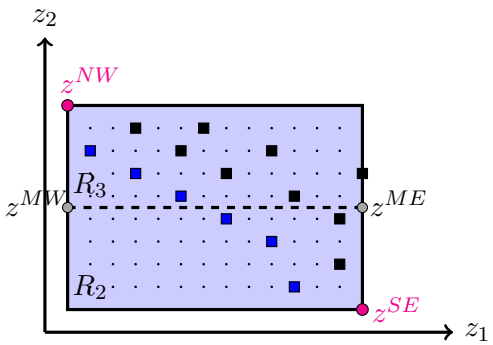
$$\operatorname*{lex\,min}_{z \in R_2} (z_1, z_2) \to \hat{z} \tag{8}$$

In our example, as this problem is feasible, we discover a new NDP, $\hat{z}$, as shown in Figure 7c. As such we now know that there is no feasible point in the region $\{z \in \mathbb{R}^2 : z_1 \leq \hat{z}_1 - \epsilon, \ z_2 \leq \hat{z}_2 - \epsilon, \}$ where ϵ is a small real number like 0.0001, and all the points in the region $\{z \in \mathbb{R}^2 : z_1 \geq \hat{z}_1, \ z_2 \geq \hat{z}_2\}$ are dominated by $\hat{z}$, thus we can remove them from our search space. More specifically, using this new information, we can **refine** the rectangles under consideration as follows:

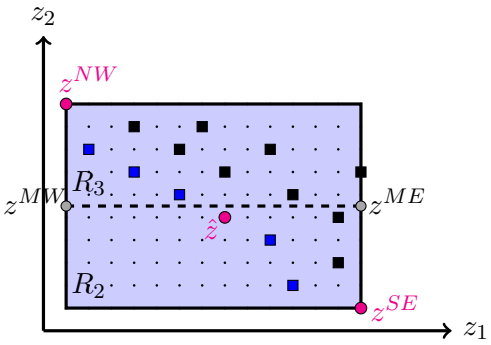
$$\bullet \ R_2 = R[\hat{z}, z^{SE}] \qquad \bullet \ R_3 = R[z^{NW}, (\hat{z}_1 - \epsilon, z_2^{MV})]$$



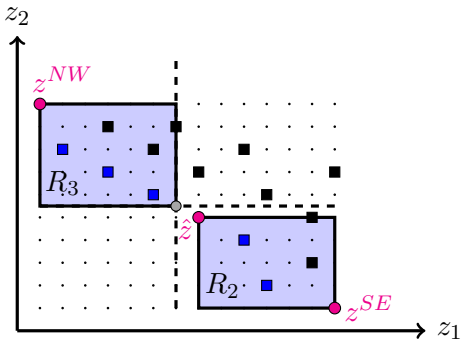
(a) Initial rectangle R_1



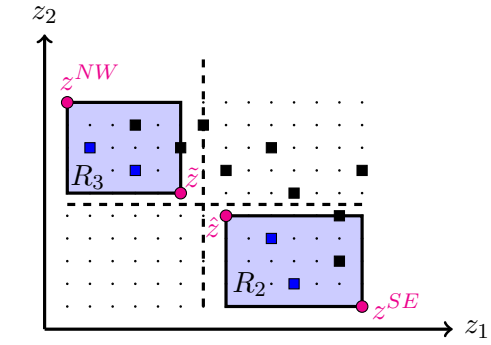
(b) Bisecting the initial rectangle R_1



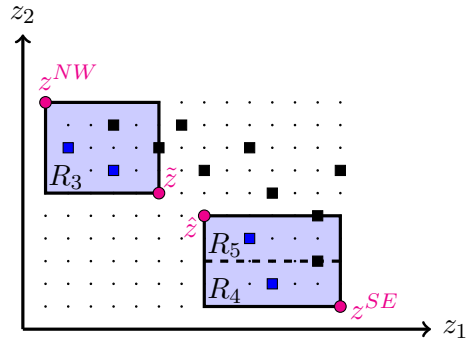
(c) Newly discovered NDP from R_2



(d) Updating the rectangles using the newly discovered NDP from R_2



(e) Newly discovered NDP from R_3 and the updated R_3



(f) Starting to process the next picked rectangle R_2

Figure 7: Illustration of the Rectangle Division Method for the hypothetical example

where we will use $\epsilon = 1$. This is demonstrated in Figure 7d, where the SE point of the updated R_3 is provided as the grey circle point.

Then, the algorithm picks the next unexplored rectangle, namely R_3, and looks for an NDP in it by solving the following lexicographical optimization problem:

$$\underset{z \in R_3}{\text{lex min}} \ (z_2, z_1) \rightarrow \tilde{z} \tag{9}$$

Note that if we are exploring a "bottom rectangle" such as R_2, we use the lexicographical order of z_1, z_2, see (8), whereas while we are exploring a "top rectangle" such as R_3, we use the lexicographical order of z_2, z_1, see (9). In our example, as this problem is feasible, we discover a new NDP, $\tilde{z}$, as shown in Figure 7e. Using this new information, we can do the following update:

$$R_3 = R[z^{NW}, \tilde{z}]$$

This is the end of the first main iteration of the algorithm. That is, in every main iteration, the algorithm picks a rectangle from a list of rectangles, divides it into two, explores one of the two rectangles and updates the two rectangles accordingly, then explores the other rectangle and updates it accordingly. In the next iteration, the algorithm picks one of the rectangles on the list, and explores it using the same procedure which we will refer to as **processing a rectangle**. Figure 7f shows the start of the processing of R_2 in our example. The algorithm will discard a rectangle if it does not contain any undiscovered NDPs, and it will stop when the list of rectangles become empty.

The full Rectangle Division Method is provided in Algorithm 2.

Algorithm 2 The Rectangle Division Method

Input: A MOP with two objectives, $\epsilon > 0$

Output: NDF as $\mathcal{Z}^N$

Create an empty list, FoundNDPs
Solve $\underset{z \in \mathcal{Z}}{\text{lex min}}\ (z_1, z_2)$, let z^{NW} be the optimal solution, FoundNDPs.add(z^{NW})
Solve $\underset{z \in \mathcal{Z}}{\text{lex min}}\ (z_2, z_1)$, let z^{SE} be the optimal solution, FoundNDPs.add(z^{SE})
Initialize the list of rectangles, Rectangles $= \{R[z^{NW}, z^{SE}]\}$
while Rectangles is not empty **do**
    Pick a rectangle $R[z^1, z^2] \in$ Rectangles ▷ Rectangle to be processed
    Remove $R[z^1, z^2]$ from Rectangles
    $R_2 = R\left[\left(z_1^1, (z_2^1 + z_2^2)/2\right), z^2\right]$ ▷ Bisect to create the bottom rectangle
    Solve $\underset{z \in R_2}{\text{lex min}}\ (z_1, z_2)$, let $\hat{z}$ be the optimal solution ▷ Look for an NDP
    if $\hat{z} \neq z^2$ **then**
        FoundNDPs.add($\hat{z}$) ▷ A new NDP is found
        Rectangles.add$\left(R[\hat{z}, z^2]\right)$ ▷ Refine the bottom rectangle
    $R_3 = R\left[z^1, \left(\hat{z}_1 - \epsilon, (z_2^1 + z_2^2)/2\right)\right]$ ▷ Create the refined top rectangle
    Solve $\underset{z \in R_3}{\text{lex min}}\ (z_2, z_1)$, let $\tilde{z}$ be the optimal solution ▷ Look for an NDP
    if $\tilde{z} \neq z^1$ **then**
        FoundNDPs.add($\tilde{z}$) ▷ A new NDP is found
        Rectangles.add$\left(R[z^1, \tilde{z}]\right)$ ▷ Refine the top rectangle
return FoundNDPs

3.3 Supernal Method

This method can be applied to general MOP problems, i.e., problems with more than two objectives. As in the Rectangle Division Method, it conducts a search of NDPs in the criterion space by dividing it into smaller regions. However, the search regions instead take the form of a shifted nonpositive orthant, which we will refer to as a **region**. More specifically, a region is represented by a single point, namely its northeast (NE) corner point and denoted by $R[\text{NE}]$. Formally, $R[z^{NE}] = \{z \in \mathbb{R}^J : z_j \leq z_j^{NE} \text{ for all } j = 1, \ldots, J\}$.

In this method, the search of an NDP in a region relies on a weighted-sum single objective optimization problem (5). For the ease of exposition we will use the following criterion space version of it:

$$\min_{z \in \mathcal{Z}} \sum_{j=1}^{J} \lambda_j z_j = \min_{z \in \mathcal{Z}} \ \lambda^\top z \tag{10}$$

However, when we are solving (10), we will be indeed solving the associated optimization problem (5). Recall that the weights must be strictly positive, i.e., $\lambda_j > 0$ for all $j = 1, \ldots, J$.

The Supernal Method is provided in Algorithm 3.

Algorithm 3 The Supernal Method

Input: A MOP, $\epsilon > 0$

Output: NDF as $\mathcal{Z}^N$

Create an empty list, FoundNDPs
Find the supernal point of MOP, z^S
Initialize the list of regions, Regions $= \{R[z^S]\}$
while Regions is not empty **do**
    Pick a region $R[z^{NE}] \in$ Regions ▷ Region to be processed
    Solve $\min\{\lambda^\top z : z \in R[z^{NE}]\}$ ▷ Look for an NDP
    if Feasible **then**
        Let z^* be an optimal solution ▷ A new NDP is found
        FoundNDPs.add(z^*)
        for each $R[\bar{z}] \in$ Regions **do** ▷ Refine any region ..
            if $z^* \in R[\bar{z}]$ **then** ▷ .. including the new NDP
                Regions.remove($R[\bar{z}]$)
                for $j = 1, \ldots, J$ **do** ▷ Split into J regions
                    Let $\bar{z}^{new} \in \mathbb{R}^J$ with $\bar{z}^{new}_j = z^*_j - 1$ and $\bar{z}^{new}_i = \bar{z}_i$ for all $i \neq j$
                    Regions.add($R[\bar{z}^{new}]$)
        if $J \geq 3$ **then**
            RemoveDominatedRegions(Regions)
return FoundNDPs

We will illustrate the concepts of this algorithm on an abstract MOP example with three objectives, also using Figure 8. Suppose that the MOP has the supernal point of $(50, 50, 50)$ which defines the initial region. We know that all the feasible images belong to this region. Then, suppose that we picked a positive weight vector λ and solved the single weighted-sum objective problem to obtain the optimal solution $(4, 4, 4)$. We know that this point belongs to NDF, so we add it to the list of discovered NDPs. Next, using this NDP, we split the initial region into three regions, one for each coordinate of the criterion space $\mathbb{R}^J$. This creates the first three branches in the tree given in Figure 8. If we branch on the z_1 component, then we obtain the region defined by the point $(3, 50, 50)$ that is obtained by modifying the first component of its parent's NE point, $(50, 50, 50)$, by subtracting $\epsilon = 1$.

Similarly, branching on the z_2 and z_3 components, we respectively obtain $R[(50, 3, 50))]$ and $R[(50, 50, 3)]$.

Before picking and processing the next region, the algorithm calls the RemoveDominatedRegions function. This function goes through the current list of regions, and removes any region that is a subset of another from that list. The reason is that the splitting method of the algorithm might create some nested regions. Consider two such nested regions: $A \subseteq B$. We call A a dominated region due to the following reason. If A is processed before B, then as the algorithm will eventually process B which includes A, the region A will be re-processed unnecessarily. Similarly, if B is processed before A, then when the algorithm later picks A to be processed, A will be re-processed unnecessarily. Therefore, in order to save some work, we can remove A from the list of regions whenever we detect $A \subseteq B$. Furthermore, if we do not remove A, then the NDPs that belong to A will be discovered multiple times as they also belong to B, in which case we would need to eliminate the duplicates from the FoundNDPs list at the end of the algorithm, which might take a significant time.

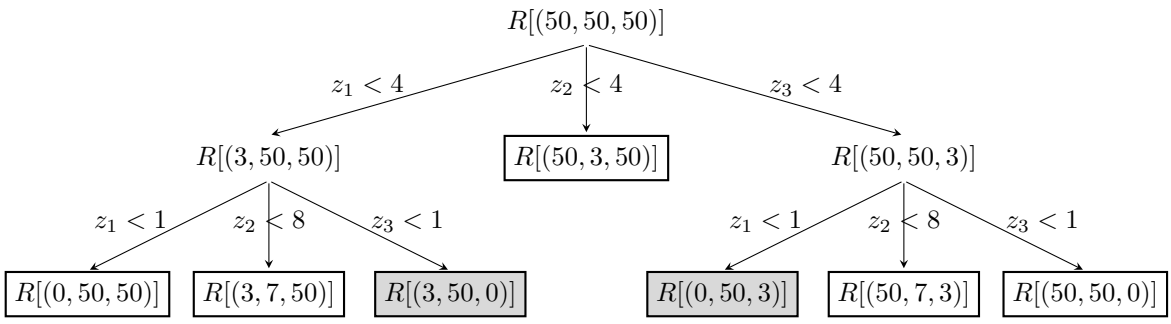


Figure 8: Algorithm 3 concepts illustrated on an abstract MOP example with three objectives and supernal point $(50, 50, 50)$, provided $\epsilon = 1$

Going back to our example, we realize that none of the newly created three regions are dominated. This is indeed always the case at the first iteration. Then, we move to the second main iteration of the algorithm where we pick a region, say $R[(3, 50, 50)]$ to be processed next. Assume that the weighted-sum problem yields the NDP $(1, 8, 1)$ from that region, which is added to FoundNDPs. As this NDP belongs to both $R[(3, 50, 50)]$ and $R[(50, 50, 3)]$, we split both of these regions into three. Now, our Regions list consists of the following regions:

$$R[(50, 3, 50)], R[(0, 50, 50)], R[(3, 7, 50)], R[(3, 50, 0)], R[(0, 50, 3)], R[(50, 7, 3)], R[(50, 50, 0)]$$

RemoveDominatedRegions function detects the following nested relationships:

$$R[(3, 50, 0)] \subseteq R[(50, 50, 0)] \text{ and } R[(0, 50, 3)] \subseteq R[(0, 50, 50)]$$

As such, it removes $R[(3, 50, 0)]$ and $R[(0, 50, 3)]$ from the Regions list. So, the algorithm will pick one of the five remaining regions to be processed in the next iteration.

4 Project Goal

In this project, you will consider the multiobjective multidimensional 0-1 knapsack problem

$$\begin{aligned} (m\text{-KP-MOP}):\ \max\ & \left\{\sum_{i=1}^{n} c_i^1 x_i, \sum_{i=1}^{n} c_i^2 x_i, \ldots, \sum_{i=1}^{n} c_i^J x_i\right\} \\ \text{s.t}\ & \sum_{i=1}^{n} a_{ik} x_i \leq b_k, && k = 1, \ldots, m \\ & x_i \in \{0, 1\}, && i = 1, \ldots, n, \end{aligned}$$

generate random instances, and experiment with the Brute-force Enumeration Method, Rectangle Division Method and Supernal Method. You can start implementing the algorithms as provided in Section 3. However, your main goal should be to develop different strategies to modify their design with the hope of improving their efficiency.

5 Random Instance Generation

In this project, you will need to measure the performance of different methods on many hard m-KP-MOP instances with various sizes. You can find and use hard instances that have been generated in the literature, or you can generate your own instances for your analysis. In the latter case, you can use Algorithm 4 to generate some random instances. **(In any case, you should have some randomness in the instance generation. While generating random numbers, you should fix the *seed* of the random number generator to the last five digits of one of your group member's student id and a multiple of that five digit number if you need more seeds.)**

Algorithm 4 m-KP-MOP random instance generation

1: Choose n, m and J.
2: Choose an upperbound $U \in \mathbb{Z}_+$, $U \geq 40$
3: Fix the seed of the random number generator
4: Generate c_i^j and a_{ik} **integer** values from the set $\{1, 2, \ldots, U\}$ randomly using uniform distribution
5: Let $b_k = \max\left\{a_{1k}, a_{2k}, ..., a_{nk}, \left\lceil \frac{1}{2} \sum_{i=1}^{n} a_{ik} \right\rceil\right\}$ for all $k = 1, \ldots, m$

For fixed n and U values (for example when $n = 10$ and $U = 40$), you can generate multiple instances, and report individual or average statistics for them. Try different n values until your computer is not able to solve the problem in a reasonable amount of time.

Also, note that we want difficult instances. Therefore, try to generate instances that are difficult to solve, i.e., ignore trivial ones that you generated, e.g., instances that can be solved in less than a second.

In the second half of the semester, we will post some challenging instances as well. Note that the instances will be provided in .txt file with the following form:

# of items (n)
Right-hand-side of the first knapsack constraint (b_1)
Right-hand-side of the second knapsack constraint (b_2)
$\vdots$
Right-hand-side of the last knapsack constraint (b_m)
The first (**minimization**) objective function coefficients ($-c^1$ vector)
The second (**minimization**) objective function coefficients ($-c^2$ vector)
$\vdots$
The last (**minimization**) objective function coefficients ($-c^J$ vector)
Constraint coefficients of the first knapsack constraint ($a_{\cdot 1}$ vector)
Constraint coefficients of the second knapsack constraint ($a_{\cdot 2}$ vector)
$\vdots$
Constraint coefficients of the last knapsack constraint ($a_{\cdot m}$ vector)

Your code is expected to read instances of this form. Also, you should return your own generated instances as .txt files of the same form.

6 Project Description

In this project, you will implement a number of algorithms that finds all nondominated points for a given multiobjective multidimensional 0-1 knapsack instance. You are expected to generate your own multiobjective multidimensional 0-1 knapsack problem instances and conduct various analyses. Before final submission, the project contains multiple **checkpoints**.

6.1 CHECKPOINT 1: Team Formation

Form your project team, **only one member** per team must submit the names and student IDs of its members. Later, we will assign each team an ID and create a page on Quercus to manage submissions.

Submit (the names and student IDs of all team members) by Jan 28, 11.59pm (EST).

6.2 CHECKPOINT 2: Instance Generator and Manual Implementation

1. Implement a random instance generator using Algorithm 4, which saves the instance in `.txt` format in the prescribed form as detailed in Section 5.
2. Create two m-KP-MOP problem instances with $n = 5$, $J \in \{2, 3\}$, $m = 1$, $U = 40$ using Algorithm 4 (one for $J = 2$ and one for $J = 3$). Set the value of `seed` equal to your team ID, i.e., `numpy.random.seed=GroupNo`. Ensure that the instances you generate have **at least 3 nondominated points** using the Brute-force Enumeration method described in Algorithm 1.
3. Manually apply the Rectangle Division Method for the instance with $J = 2$, and illustrate it step by step.
4. Manually apply the Supernal Method for both of the instances, and illustrate it step by step.

The structure of the Python script for the Brute-Force Enumeration method is detailed in the subsequent subsections, along with the expected output files. Process the NDP output file to determine if the generated instance has 3 or more nondominated points and manually solve them as mentioned in step 3 and 4 above.

6.2.1 Python Code for Brute-force Enumeration

[IMPORTANT] The algorithm *must* be coded in Python. Submit a `.py` script that is called `SolveKnapsack_GroupNo.py`. We will provide group numbers after Checkpoint 1. Your script should contain a function `SolveKnapsack` which takes two arguments,

- `inputfile`: Absolute path to the instance file which needs to be solved. The format of the instance file should follow the structure detailed in Section 5.
- `method`: An integer argument with a default value of 1. Over the course of the project, we will implement various algorithms and this argument will help us select which one we would like to execute.

The `SolveKnapsack_GroupNo.py` script should be structured as follows.

```
# imports
import time
import copy
import collections
import queue as Q
import numpy as np
import pandas as pd
import scipy as sp

```

```python

def SolveKnapsack(filename, method=1):
  # Dummy group number. Should be replaced by your number
  groupNo = 1212
  methodName = ''

  if method == 1:
    methodName = "BF"
    # TODO: Read and solve an instance via Brute-Force method

  # Output result
  ndp_filename = f'{methodname}_NDP_{groupNo}.txt'
  summary_filename = f'{methodname}_SUMMARY_{groupNo}.txt'

  # TODO: Export NDP and Summary files

  return
```

Please ensure that the `SolveKnapsack_GroupNo.py` is self-contained, i.e., it should not import from other python package/module. One can define other helper functions for better code organization. Take a look at the below code snippet for an example.

```python
'''
imports
'''

# Not allowed
from my_pkg.my_mod import my_custom_fn

# Allowed
def my_helper_1():
  '''
  Can contain helper functions
  '''

def SolveKnapsack(filename, method):
  '''
  Logic
  '''

# Not allowed
time.sleep(1000)
# Other random commands
```

Any code submission not meeting this criterion will be heavily penalized while grading the final project submission. So, please make sure at this checkpoint stage that your code would be fine.

6.2.2 Output Files

The `SolveKnapsack(inputfile, method)` function should output `methodName_NDP_GroupNo.txt` and `methodName_SUMMARY_GroupNo.txt` in the same folder as the `SolveKnapsack_GroupNo.py` script, where `methodname` will be set to BF. To obtain the current working directory use the `os` module.

```
import os
curr_dir = os.getcwd()
```

NDP file: The `methodname_NDP_GroupNo.txt` files should contain the non-dominated points. Each row corresponds to an NDP where the first number shows the first objective value, the second one shows the second objective value, and so on. **Moreover, the NDPs must be sorted in a lexicographical decreasing order, i.e., first sorted in terms of the first objective value from the highest to the lowest, then in terms of the second objective value, and so on**. Since we are assuming that we have a negative cost, it will not have any strictly positive points. Assuming the `ndp_array` contains the NumPy array of non-dominated points, the below code snippet illustrates how to save it in a text file.

```
import numpy as np
import os

curr_dir = os.getcwd() + '/'
# Dummy NDP array of size (6, 3)
ndp_array = np.array([[-33989, -27089, -25778],
                      [-34021, -26358, -25645],
                      [-33988, -27166, -25088],
                      [-26293, -27623, -31753],
                      [-26809, -34008, -26540],
                      [-26448, -34023, -26294]])

# Note: You must set delimiter to '\t' and newline to '\n'. Otherwise,
    points will be deducted.
np.savetxt(curr_dir + "sample.txt",
           ndp_array,
           delimiter='\t',
           newline='\n')
```

Here the shape of the array is (6,3) and we are using the `np.savetxt` function to write it to a text file. Note that the `delimiter='\t'` and `newline='\n'`. As a sanity check, ensure that you are correctly able to read the file using `loadtxt` as follows:

```
np.loadtxt(curr_dir + "sample.txt", delimiter='\t')
```

The submissions for which we are not able to read its output using `np.loadtxt(path_to_output_file, delimiter='\t')` will be heavily penalized while grading the final project submission. So, please make sure at this checkpoint stage that your output format would be fine.

Summary file: Finally, readers are required to export some summarized results in .txt format. This must be a one-dimensional NumPy array of the form: `[Solution time measured in seconds, Number of obtained NDPs, 0]`. Your python code should export such NumPy array in txt format with the name `methodName_SUMMARY_GroupNo.txt`. Note that you need to export the summary array to a text file using the `np.savetxt` function and `delimiter='\t'` and `newline='\n'`. Again, the submissions for which we are not able to read its output using `np.loadtxt(path_to_output_file, delimiter='\t')` will be heavily penalized. So, please make sure at this checkpoint stage that your summary format would be fine.

Submit the following items (along with the names and student IDs of all team members) by Feb 18, 11.59pm (EST):

1. Instance generator script
2. Generated instances
3. Brute-force Enumeration script with the name `SolveKnapsack_GroupNo.py`
4. Manual implementation files

6.3 CHECKPOINT 3: Implement Rectangle Division Method

Generate random instances using the instance generator and implement the Rectangle Division Method to solve them. The goal of this checkpoint is to correctly enumerate all the nondominated points without worrying about algorithm efficiency in terms of time.

6.3.1 Python Code for Rectangle Division Method

[IMPORTANT] Update the `SolveKnapsack_GroupNo.py` file with the implementation of the Rectangle Division Method. The `SolveKnapsack_GroupNo.py` script should now look as follows.

```
# Python imports
import time
import copy
import collections
import queue as Q
import numpy as np
import pandas as pd
import scipy as sp

import gurobipy as gp
from gurobipy import GRB


def SolveKnapsack(filename, method):
```

```
  # Dummy group number. Should be replaced by your number
  groupNo = 1212
  methodName = ''

  if method == 1:
    methodName = "BF"
    # TODO: Read and solve an instance via Brute-Force method

  elif method == 2:
    methodName = "RDM"
    # TODO: Read and solve an instance via Rectangle Divison Method (RDM)

  # Output result
  ndp_filename = f'{methodname}_NDP_{groupNo}.txt'
  summary_filename = f'{methodname}_SUMMARY_{groupNo}.txt'

  # TODO: Export NDP and Summary files

  return
```

6.3.2 Output Files

The `SolveKnapsack(inputfile, method)` function should output `methodName_NDP_GroupNo.txt` and `methodName_SUMMARY_GroupNo.txt` in the same folder as the `SolveKnapsack_GroupNo.py` script, where `methodname` will be set to RDM.

NDP file The `methodName_NDP_GroupNo.txt` will exactly follow the same structure as Checkpoint 2.

Summary file The `methodName_SUMMARY_GroupNo.txt` must contain a one-dimensional NumPy array of the form: `[Solution time measured in seconds, Number of obtained NDPs, Number of Boxes generated OR Number of Regions generated]`. Unlike Checkpoint 1, the last element in this case tracks the number of boxes generated.

Submit the the following item (along with the names and student IDs of all team members) by March 25, 11.59pm:

1. Brute-force Enumeration script with the name `SolveKnapsack_GroupNo.py`.

6.4 Project

In this project, you will implement a number of algorithms that finds all nondominated points for a given multiobjective multidimensional 0-1 knapsack instance. Moreover, you are going to do various comparative and exploratory analyses using your algorithms.

You should answer following questions:

1. **Implementation**: Write a Python code that reads a given instance, and finds all nondominated points by implementing the Brute-Force Enumeration, Rectangle Division and Supernal Methods.

2. **Efficiency**: In this question you are expected to compare performances of the Brute-force Enumeration, Rectangle Division and Supernal Methods for different instances with $J = 2$ by varying n and m. For different instances with $J > 2$, varying n, J and m apply the Supernal Method and analyze its performance with respect to **solution times**.

3. **Improvements**: In this question you are expected to try different approaches for algorithmic improvements. Potential improvements for the Brute-force Enumeration Method might be trying different variable ordering or elimination methods to quickly eliminate dominated points. For the Rectangle Division Method potential improvements can be dividing the space differently (e.g., not dividing into two equal parts), changing the order of the lexicographical optimization and trying different rectangle selection rules (e.g., the largest area, the smallest area, bottom first, top first, random etc.). For the Supernal Method, you can explore different λ values and different region selection rules. Lastly, for all the methods, you can benefit from more efficient sorting methods, domination checks, etc.

4. **Complexity Analysis**: Discuss the theoretical complexity (to the best of your ability) of the following algorithms that you implement and/or some smaller tasks involved in these algorithms:

   - Brute-force Enumeration Method
   - Rectangle Division Method
   - Supernal Method

5. **Memory issues**: Mention whether you have encountered any memory issues in your experiments. Comment on your expectation for the memory requirement for all the methods.

Submit the deliverables detailed in the subsequent section (along with the names and student IDs of all team members) by April 15, 11.59pm.

7 Deliverables

The project has three deliverables:

- Project report in PDF (no other file type will be accepted), prepared in LaTeX.
- Zip file of all your code and instances.
- Attribution tables. (Each team member will submit an individual table.)

7.1 Report

Your written report should contain everything about your algorithm implementation and comparison process. There is no fixed template for the report. However, if you are looking for suggestions, you may use templates from arXiv, INFORMS Journal on Computing or any top conference in your area of research. There is no minimum or maximum page limit; just write enough to thoroughly describe your process/experience, and *no more*. Remember the report is also graded on clarity and conciseness. At the very least, your report should contain the following information (the section outline is provided as an example):

Introduction Explain the advantages of using introduced methods compared to Brute-force Enumeration and why we need them. Introduce the steps you take in implementing the algorithms. Explain what criteria you use to compare the performances of different algorithms and why you are concerned with those criteria.

Algorithms Provide details about each of the three algorithms you implemented, including pseudo-codes and complexity analysis. Especially, provide any implementation details meant to improve the efficiency of certain algorithmic steps.

Data Generation Explain in detail which instances you used to test your algorithms on and how you generated your own instances.

Algorithm Comparison Conduct required experiments and analyses that are specified in Section 6.4. Provide a detailed explanation and interpretation of the results you obtained from your analyses. Use plots and tables whenever appropriate. Make sure the plots and tables you use are clear and easy to read. (You can directly form the tables in LaTeX, while you can use matplotlib to prepare your plots, save them as pdf and include them in LaTeX. Do **not** use screenshots.)

Conclusion Make your final recommendation for the best algorithm and discuss anything interesting you observed during the process. Also, provide your final comments about your work.

7.2 Code

[IMPORTANT] All algorithms *must* be coded in Python. Submit a `.py` script called `SolveKnapsack_GroupNo.py`. Your script should contain a function `SolveKnapsack` which takes two arguments:

- `inputfile`: Absolute path to the instance file which needs to be solved. The format of the instance file should follow the structure detailed in Section 5.
- `method`: An integer argument between 1 and 5. Specifically,
  - 1 for the Brute-Force Enumeration,
  - 2 for the Rectangle Division Method,
  - 3 for the Supernal Method,
  - 4 for the competition method for solving instances with $J = 2$,
  - 5 for the competition method for solving instances with $J = 3$.

A call to `SolveKnapsack(inputfile, method)` should read an instance in `inputfile`, apply `method` to solve it, and provide the relevant output. Note that for `method` 1, 2 and 3 the evaluation would be focused on correctness. As a result, a naive implementation would be fine. However, for `method` 4 and 5, the focus would be on execution time (along with correctness). Hence you are encouraged to make appropriate modifications to improve the running time of your script. The `SolveKnapsack_GroupNo.py` script should be structured as follows.

```
# Python imports
import time
import copy
import collections
import queue as Q
import numpy as np
import pandas as pd
import scipy as sp

import gurobipy as gp
from gurobipy import GRB


def SolveKnapsack(filename, method):
  # Dummy group number. Should be replaced by your number
  groupNo = 1212
  methodName = ''

  if method == 1:
    methodName = "BF"
    # TODO: Read and solve an instance via Brute-Force method
```

```
22
23    elif method == 2:
24      methodName = "RDM"
25      # TODO: Read and solve an instance via Rectangle Divison Method (RDM)
26
27    elif method == 3:
28      methodName = "SPM"
29      # TODO: Read and solve an instance via Supernal Method (SPM)
30
31    elif method == 4:
32      methodName = "COMP_2D"
33      # TODO: Read and solve an instance via RDM or SPM designed for
        competition with J=2
34
35    elif method == 5:
36      methodName = "COMP_3D"
37      # TODO: Read and solve an instance via RDM or SPM designed for
        competition with J=3
38
39
40    # Output result
41    ndp_filename = f'{methodname}_NDP_{groupNo}.txt'
42    summary_filename = f'{methodname}_SUMMARY_{groupNo}.txt'
43
44    # TODO: Export NDP and Summary files
45
46    return
```

As previously stated, the `SolveKnapsack_GroupNo.py` should be self-contained. Your submission will be evaluated on our end in the following manner.

```
1 import SolveKnapsack_GroupNo as solver
2
3 filename = 'path/to/file'
4 methods = [1, 2, 3, 4, 5]
5
6 # The method argument can take integer values from 1 to 5
7 for method in methods:
8   solver.SolveKnapsack(filename, method)
```

7.3 Output Files

The `SolveKnapsack(inputfile, method)` function should output `methodName_NDP_GroupNo.txt` and `methodName_SUMMARY_GroupNo.txt` in the same folder as the `SolveKnapsack_GroupNo.py` script, where `methodname` can be BF, RDM, SPM, COMP_2D or COMP_3D and `GroupNo` is your team's group number.

NDP file: The `methodName_NDP_GroupNo.txt` will follow the same structure as Checkpoint 3.

Summary file: The `methodName_SUMMARY_GroupNo.txt` will follow the same structure as Checkpoint 3.

8 Grading Scheme

Table 1: Grading Scheme

Item	Weight
Report	40%
Algorithm code	30%
Correctness of the results	15%
Efficiency	15%

Report: You will be assessed on the thoroughness of each of the report sections described in Section 7.1, as well as on the organization, quality and appropriateness of figures/tables.

Algorithm code: You will be assessed on correctness and style (i.e., formatting, appropriate use of functions, etc.).

Correctness of the results: Correctness of your algorithms will be verified on our own instances.

Efficiency: Here you will be provided a set of instances with $J = 2$ and $J = 3$. If your algorithm runs in an "expected" range of efficiency, you will receive 8 points. Additionally, 7 points will be distributed based on competition. The team with the fastest running time will get all 7 points, and everyone else will receive points in the range $[1, 7]$ proportional to computation time. Specifically, if f is the fastest computation time, computation time t earns the following points: $\max\{1, 7 \times (f/t)\}$.

9 Teams

Teams will (mostly) consist of four people and must be finalized in Checkpoint 1. Depending on how many people are in the class, some three-person teams might be necessary. Teams should not have more than four people. There will be no difference in grading three-person teams vs four-person teams. Teams are strongly recommended to have at least one person with a good knowledge in Python.

References

[1] Arnaud Fréville. The multidimensional 0–1 knapsack problem: An overview. *European J. Oper. Res.*, 155(1):1–21, 2004.