Algebra for Machine Learning and Stochastic Programming II

Yuchen Ge

September 2022

Contents

Abstract

This article introduces cost-space scenario clustering (CSSC) method in Stochastic Optimization and how Gröbner and Graver Basis is helpful to reduce the computational difficulty in the method.

1 Problem Statement

We consider the following stochastic optimization problem:

$$\min_{x\in X\subseteq\mathbb{R}^n}\left\{f(x):=\frac{1}{N}\sum_{i=1}^{N}F\left(x,\xi_i\right)\right\}, \tag{1}$$

where $\xi_1,\ldots,\xi_N$ are equiprobable scenarios taking values in $\mathbb{R}^d$ and $F\left(x,\xi_i\right)$ represents the cost associated to the decisions $x\in X$ in scenario ξ_i. The optimal solution of this problem is denoted by x^* and its optimal value by v^*. The problem is formulated using equiprobable scenarios for simplicity; the method introduced can be easily generalized to scenarios with different probabilities.

The cost function F may be given explicitly (if the problem is one-stage), or may be itself the result of a second-stage optimization problem. So

1. **In the first case, if the problem is one-stage, it can be viewed as a second-stage problem.**

2. In this latter case, which is the framework of two-stage stochastic programming, it typically takes the following form:

$$F\left(x,\xi_i\right)=\min_{y\in Y(x,\xi_i)}g\left(x,y,\xi_i\right),$$

1.1 Method of cost-space scenario clustering

The subsection is based on [1].

Suppose that problem (1) cannot be solved as it is, so we need to build from it an approximate problem composed of K scenarios with $K\ll N$. There are two broad ways to generate those scenarios:

1. they may be picked directly in the original set $\{\xi_1,\ldots,\xi_N\}$
2. they may be completely new scenarios that do not exist in the original set.

Consider for now that this set has been computed, and let us denoted it by $\left\{\widetilde{\xi}_1,\ldots,\widetilde{\xi}_K\right\}$ with the corresponding probabilities $\{p_1,\ldots,p_K\}$. The approximate problem takes the form:

$$\min_{x\in X\subseteq\mathbb{R}^n}\left\{\widetilde{f}(x):=\sum_{k=1}^{K}p_kF(x,\widetilde{\xi}_k)\right\}$$

and its optimal solution is denoted by $\widetilde{x}^*$.

Then to generate these scenarios, we measure the distance (proximity) of two scenarios by means of the space of cost values ($\mathbb{R}$).

First, let us first decompose the implementation error as follows:

$$\begin{aligned}\left|f\left(\widetilde{x}^*\right)-v^*\right|=\left|f\left(\widetilde{x}^*\right)-f\left(x^*\right)\right|&=\left|f\left(\widetilde{x}^*\right)-\widetilde{f}\left(\widetilde{x}^*\right)+\widetilde{f}\left(\widetilde{x}^*\right)-f\left(x^*\right)\right|\\&\leq\left|f\left(\widetilde{x}^*\right)-\widetilde{f}\left(\widetilde{x}^*\right)\right|+\left|\widetilde{f}\left(\widetilde{x}^*\right)-f\left(x^*\right)\right|\end{aligned} \tag{2}$$

The second term can be further bounded by:

$$\begin{aligned}\left|\tilde{f}\left(\widetilde{x}^*\right)-f\left(x^*\right)\right|&=\max\left\{\widetilde{f}\left(\widetilde{x}^*\right)-f\left(x^*\right),f\left(x^*\right)-\widetilde{f}\left(\widetilde{x}^*\right)\right\}\\&\leq\max\left\{\widetilde{f}\left(x^*\right)-f\left(x^*\right),f\left(\widetilde{x}^*\right)-\widetilde{f}\left(\widetilde{x}^*\right)\right\}\\&\leq\max\left\{\left|\widetilde{f}\left(x^*\right)-f\left(x^*\right)\right|,\left|f\left(\widetilde{x}^*\right)-\widetilde{f}\left(\widetilde{x}^*\right)\right|\right\}\\&=\max_{x\in\{x^*,\widetilde{x}^*\}}|\widetilde{f}(x)-f(x)|\end{aligned} \tag{3}$$

Finally, by combining (2) and (3) we can bound the implementation error as follows. Let $\widetilde{X} \subseteq X$ be any feasible set such that $\{x^*, \widetilde{x}^*\} \subset \widetilde{X}$ and we have

$$\begin{aligned}
|f(\widetilde{x}^*) - v^*| &\leq 2 \max_{x \in \{x^*, \widetilde{x}^*\}} |\tilde{f}(x) - f(x)| \\
&\leq 2 \max_{x \in \widetilde{X}} |\tilde{f}(x) - f(x)| \\
&= 2 \max_{x \in \widetilde{X}} \left| \frac{1}{N} \sum_{i=1}^{N} F(x, \xi_i) - \sum_{k=1}^{K} p_k F\left(x, \widetilde{\xi}_k\right) \right| \\
&= 2 \max_{x \in \widetilde{X}} \left| \frac{1}{N} \sum_{k=1}^{K} \sum_{i \in C_k} F(x, \xi_i) - \sum_{k=1}^{K} \frac{|C_k|}{N} F\left(x, \widetilde{\xi}_k\right) \right| \\
&= 2 \max_{x \in \widetilde{X}} \left| \sum_{k=1}^{K} \frac{|C_k|}{N} \left(\frac{1}{|C_k|} \sum_{i \in C_k} F(x, \xi_i) - F\left(x, \widetilde{\xi}_k\right) \right) \right| \\
&\leq 2 \sum_{k=1}^{K} p_k \sup_{x \in \widetilde{X}} \left| \frac{1}{|C_k|} \sum_{i \in C_k} F(x, \xi_i) - F\left(x, \widetilde{\xi}_k\right) \right| \\
&=: 2 \sum_{k=1}^{K} p_k D(C_k)
\end{aligned} \tag{4}$$

The quantity $D(C_k)$ can be seen as the discrepancy of the cluster C_k . It measures how much the cost function $F(x, \widetilde{\xi}_k)$ of its representative scenario $\widetilde{\xi}_k$ matches the average cost values of the whole cluster C_k over the feasible set $\tilde{X}$. we then approximate $D(C_k)$ by:

$$D(C_k) \simeq \left| \frac{1}{|C_k|} \sum_{i \in C_k} F(x_k^*, \xi_i) - F\left(x_k^*, \widetilde{\xi}_k\right) \right|$$

where $x_k^* \in \underset{x \in X}{\operatorname{argmin}} F(x, \widetilde{\xi}_k)$. The next subsection describes the algorithm in more details and analyze its computational cost. Then we can state the algorithm as follows:

Algorithm 1.1. 1. Compute the opportunity-cost matrix $\boldsymbol{V} = (V_{i,j})$ where $V_{i,j} = F(x_i^*, \xi_j)$ and $x_i^* \in \underset{x \in X}{\operatorname{argmin}} F(x, \xi_i)$.
2. Find a partition of the set $\{1, \ldots, N\}$ into K clusters $C_1, \ldots, C_K$ and their representatives $r_1 \in C_1, \ldots, r_K \in C_K$ such that the following clustering discrepancy is minimized:

$$\sum_{k=1}^{K} p_k \left| V_{r_k, r_k} - \frac{1}{|C_k|} \sum_{j \in C_k} V_{r_k, j} \right|,$$

where $p_k = \frac{|C_k|}{N}$.

The second step is equivalent to

$$\begin{aligned}
\min \quad & \tfrac{1}{N} \textstyle\sum_{i=1}^{N} t_i \\
\text{s.t.} \quad & t_j \geq \textstyle\sum_{i=1}^{N} x_{ij} V_{j,i} - \sum_{i=1}^{N} x_{ij} V_{j,j}, && \forall j \in \{1, \ldots, N\}; \\
& t_j \geq \textstyle\sum_{i=1}^{N} x_{ij} V_{j,j} - \sum_{i=1}^{N} x_{ij} V_{j,i}, && \forall i \in \{1, \ldots, N\}; \\
& x_{ij} \leq u_j, \quad x_{jj} = u_j && \forall (i,j) \in \{1, \ldots, N\}^2; \\
& \textstyle\sum_{j=1}^{N} x_{ij} = 1, \quad \sum_{j=1}^{N} u_j = K && \forall i \in \{1, \ldots, N\}.
\end{aligned} \tag{5}$$

It should be noted that this means that two scenarios with very different cost values may still be included in the same cluster if there exists a third scenario whose value provides a mid-ground between them.

2 Prerequisite on Gröbner Basis and Graver Basis

For an integer programming problem:

$$(IP)_{c,b} : \min \{cx : Ax = b \text{ and } x \in N^n\},$$

we have Grobner Basis $\mathcal{G}_{c,A}$ and Graver Basis $\mathcal{GR}_A$, where the subscript represents what necessarily determines the basis.

First we recall that the reduced Gröbner Basis and test set are related as follows.

$$(x^{\alpha_i} - x^{\beta_i}) \leftrightarrow (\alpha_i - \beta_i)$$

And by choosing appropriate cost c and monomial order $>$, we can get all monomial orders in the form of the composite order $>_c$, for example, lexicographic order. See for the details in the proof of theorem 10 in [2].

3 Improvement in computation by Gröbner Basis

When ξ_i varies,

$$F(x, \xi_i)$$

also varies. So to best diminish the computational cost, we study the relationships between the Gröbner Basis of $(IP)_{c,b} : \min \{cx : Ax = b \text{ and } x \in N^n\}$ as b, c varies. Generally, there's no relationship between them.

It is worth to be noted that if the matrix A changes when ξ_i varies, we couldn't make use of the relationships between the Gröbner Basis. This is because they are unpredictable.

Example Let

$$A = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$$

Then we have the toric ideal of A, denoted by $I_A = \{x^u - x^v : u, v \in \mathbb{Z}^n_{\geq 0}, Au = Av\}$, is $\left\langle x-y, y-z, z-x \right\rangle$. So let $>_c$= lexicographic order, we have $\mathcal{G}_{c,A} = \{(1,0,1), (0,1,1)\}$ and $>_{c'}$= lexicographic order w.r.t.$z > x > y$ we have $\mathcal{G}_{c',A} = \{(1,1,0), (0,1,1)\} \neq \mathcal{G}_{c,A}$. The results above are proved by the following **Sage Codes**.

```
sage: A=matrix([1,1,1])
sage: IA = ToricIdeal(A)
sage: IA
Ideal (z0 - z1, -z1 + z2) of Multivariate Polynomial Ring in z0, z1, z2 over Rational Field

sage: I = Ideal([x-y,y-z,z-x])
sage: I.groebner_basis()
[x - z, y - z]
```

Actually, no relationships can be found when b, c are varying since the change can be random. However, we can significantly reduce the time of computation by means of the following route.

Algorithm 3.1. (Compute the Gröbner Basis fixing each ξ_j)
Input: matrix A and c of $(IP)_{c,b}$
Output: a test set $\mathcal{T}_c$ for $(IP)_{c,b}$
1. Compute a small generating set of Ker(A). (**Algorithm 6.13 in [2]**)
2. Apply the bunchberger's algorithm to compute the the Gröbner Basis of ξ_j

So from the algorithm, only A and the required ingredient of the scenario ξ_j is needed as an input. And the computational cost is greatly reduced since we don't need to compute the Gröbner Basis of an ideal spanned by a large set any more.

4 Improvement in computation by Graver Basis

First we shall recall some new facts about graver basis of $(IP)_{c,b}$, which is denoted by $\mathcal{GR}_A$.

Let $\mathcal{G}_{A,c}$ be the reduce Gröbner basis of $(IP)_{c,b}$. Then

$$\bigcup_{c\in\mathbb{Z}^n} \mathcal{G}_{A,c} \subseteq \mathcal{GR}_A.$$

Moreover, for the relationship between the Graver Basis (Universal Gröbner Basis) of the SIP

$$\min\left\{c^Tx+\sum_{i=1}^{N} p_iq_i^Ty_i : Ax=b, x\in\mathbb{Z}_+^m, Tx+Wy_i=h_i, y_i\in\mathbb{Z}_+^n, i=1,\dots,N\right\} \tag{6}$$

and that of the IP

$$\min\left\{c^Tx+p_1q_1^Ty : Ax=b, x\in\mathbb{Z}_+^m, Tx+Wy=h, y\in\mathbb{Z}_+^n\right\} \tag{7}$$

we assert

Theorem 4.1. *(new/true) Denote the Graver Basis of (8) by $\mathcal{GR}_N$, and that of (9) by $\mathcal{GR}_1$, $i=1,2,...,N$. Then if* $\mathrm{Ker}_{\mathbb{R}}(A)=\{0\}$*, we have*

$$\mathcal{GR}_N=\{(0,0,...,v_i,...,0) : (0,v_i)\in\mathcal{GR}_1, i=1,2,...,n\}$$

and

Theorem 4.2. *(new) Denote all building blocks of Graver Basis of (7) by $\mathcal{H}_N$. Then we have for $N\geq 2$*

$$\mathcal{H}_N=\mathcal{H}_2$$

From above theorems and propositions, we have two new algorithms.

Algorithm 4.3. (Augmentation Algorithm for IP)
Input: a feasible solution z_0 to $(IP)_{c,b}$, a universal test set $\mathcal{T}$ for $(IP)_{c,b}$
Output: an optimal point $z_{\min}$ of $(IP)_{c,b}$
while there is $t\in\mathcal{T}$ with $c^Tt>0$ such that z_0-t is feasible do

$$z_0:=z_0-t$$

return:z_0

Algorithm 4.4. (Construction of Universal Test Set for (6) under the Condition of Theorem 1)
Input: a universal test set $\mathcal{T}$ for (7)
Output: a universal test set $\mathcal{T}$ for (6)
1. Construct the set as in Theorem 1.

Algorithm 4.5. (Augmentation Algorithm for (6))
Input: a feasible solution z_0 to (7) , a universal test set $\mathcal{T}$ for (6)
Output: an optimal point $z_{\min}$ of $(IP)_{c,b}$
1. Compute the building blocks for (6) when $N=2$.
2. Apply building blocks to find an optimum of (6) (**Lemma 9 and Lemma 10 in [2]**)

5 Some Examples

5.1 Example I

For the following two-stage problem

$$\begin{aligned}
\min & \frac{1}{4}\sum_{i=1}^{4}\left(2t_1^i+3t_2^i-y_1^i\xi_i^1-y_2^i\xi_i^2\right)\\
\text{s.t. } & x+y_1^i\xi_i^1-t_1^i\le 0 \quad i\in\{1,2,3,4\};\\
& x+y_1^i\xi_i^1+t_1^i\ge 0 \quad i\in\{1,2,3,4\};\\
& x-y_2^i\xi_i^2-t_2^i\le 0 \quad i\in\{1,2,3,4\};\\
& x-y_2^i\xi_i^2+t_2^i\ge 0 \quad i\in\{1,2,3,4\};\\
& x\in\mathbb{R}, t_1^i\ge 0, t_2^i\ge 0, y_1^i\in\{-1,1\}, y_2^i\in\{-1,1\} \quad i\in\{1,2,3,4\}.
\end{aligned}$$

This problem has one decision variable at stage 0 (x) , four decision variables at stage 1 (t_1, t_2, y_1, y_2) , two random parameters $\left(\xi_i=\left(\xi_i^1,\xi_i^2\right)\right)$, and four scenarios: $\xi_1=(0,0.9), \xi_2=(0,-1), \xi_3=(1.1,0)$ and $\xi_4=(-1,0)$. Its (unique) optimal solution is $x^*=0$ with objective value $v^*=1.475$.

From this example, we can see that in the notation of

$$\min_{x\in X\subseteq\mathbb{R}^n}\left\{f(x):=\frac{1}{N}\sum_{i=1}^{N}F\left(x,\xi_i\right)\right\},$$

we can write it in the form of

$$\begin{aligned}
F(x,t^i,y^i,\xi_i)=\min & \left(2t_1^i+3t_2^i-y_1^i\xi_i^1-y_2^i\xi_i^2\right)\\
\text{s.t. } & \begin{pmatrix}1\\1\\1\\1\end{pmatrix}x+\begin{pmatrix}\xi_i^1 & \\ \xi_i^1 & \\ & -\xi_i^2\\ & -\xi_i^2\end{pmatrix}y^i+\begin{pmatrix}-1 & \\ 1 & \\ & -1\\ & 1\end{pmatrix}t^i+\ldots=0
\end{aligned}$$

where $t^i=\begin{pmatrix}t_1^i\\t_2^i\end{pmatrix}$, $y^i=\begin{pmatrix}y_1^i\\y_2^i\end{pmatrix}$ and '...' denotes some slack variables to make the inequalities become equalities.

From the above form, we can see that the scenario ξ_i is in the matrix A if we insist the IP notation. So improvement by Gröbner Basis and Graver Basis is not adaptable.

5.2 Example II

Consider a stochastic network design problem, where a number of commodities are to be transported across a network that has to be designed before the demand of these commodities is known. For a complete directed graph $G=(V,A)$, across which commodities from a set C must be transported. Each arc $a\in A$, pointing from $a(0)\in V$ to $a(1)\in V$, has a total capacity u_a if it has been opened for a fixed cost c_a . The cost of transporting one unit of commodity $c\in C$ across arc $a\in A$ is denoted by q_{ac} . We denote the demand for commodity $c\in C$ at vertex $v\in V$ under scenario i by $d_{v,c}^i$.

The first stage decision of opening the arc $a\in A$ is denoted by x_a , which can take two values: 0 if the arc is closed and 1 if it is open. The second stage decision y_{ac}^i is the number of units of commodity $c\in C$ transported across arc $a\in A$ under scenario i We restrict it to be integral. This problem is solved using the following two-stage stochastic formulation:

$$
\begin{aligned}
\min & \sum_{a\in A} c_a x_a + \frac{1}{N}\sum_{i=1}^{N}\sum_{c\in C}\sum_{a\in A} q_{ac} y_{ac}^i \\
\text{s.t.} & \sum_{\substack{a\in A\\ a(0)=v}} y_{ac}^i - \sum_{\substack{a\in A\\ a(1)=v}} y_{ac}^i = d_{v,c}^i \quad \forall (v,c,i) \in V\times C\times\{1,\dots,N\} \\
& \sum_{c\in C} y_{ac}^i \le u_a x_a \quad \forall (a,i)\in A\times\{1,\dots,N\} \\
& x_a \in \{0,1\}, y_{ac}^i \in \mathbb{Z}^+
\end{aligned}
$$

Similar to example I, we can transform it into the matrix version. And we can easily see that improvement by Gröbner Basis and Graver Basis is adaptable.

5.3 Example III

Consider the facility location problem formulated as follows:

$$
\begin{aligned}
\min & \sum_{f\in F} c_f x_f + \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{c\in C}\sum_{f\in F} q_{cf} y_{cf}^i + \sum_{f\in F} b_f z_f^i\right) \\
\text{s.t.} & \sum_{f\in F} x_f \le v \\
& \sum_{c\in C} d_{cf} y_{cf}^i \le u x_f + z_f^i \quad \forall (f,i)\in F\times\{1,\dots,N\} \\
& z_f^i \le M x_f \quad \forall (f,i)\in F\times\{1,\dots,N\} \\
& \sum_{f\in F} y_{cf}^i = h_c^i \quad \forall (c,i)\in C\times\{1,\dots,N\} \\
& x_f\in\{0,1\}, y_{cf}^i\in\{0,1\}, z_f^i\in[0,\infty) \quad \forall (c,f,i)\in C\times F\times\{1,\dots,N\}
\end{aligned}
$$

where still the i specifies the scenario.

As we can easily recognize, this is a two-stage SMIP with the first-stage variable combinatorial and the second-stage mixed integral.

Here is an easy way to tell whether we can adapt improvement by Gröbner Basis and Graver Basis. By concentrating on the scenario-dependent constants, for example, h_c^i (constants have subscriptions/superscriptions i), we can see that it's not in the matrix A of the second-stage problem, if we write the second-stage problem in the matrix notation as in example I. So for this problem computational cost can be reduced.

However, the second-stage problem is a mixed-integer problem. So we need further generalizations of the original method to the SMIP case.

6 Numerical Experiment

The central idea of all the algorithms above is to extract some common steps needed without the dissipate of computational cost. And the realization of the algorithm is only dependent on the common package, which is beneficial to its compatibility.

Recall the famous example in [3]

$$
\begin{aligned}
\min \left\{ 35x_1 + 40x_2 + \frac{1}{N}\sum_{\nu=1}^{N} 16y_1^\nu + 19y_2^\nu + 47y_3^\nu + 54y_4^\nu \right\} \\
\text{s.t. } & x_1 + y_1^\nu + y_3^\nu \geq \xi_1^\nu \\
& x_2 + y_2^\nu + y_4^\nu \geq \xi_2^\nu \\
& 2y_1^\nu + y_2^\nu \leq \xi_3^\nu \\
& y_1^\nu + 2y_2^\nu \leq \xi_4^\nu, \\
& x_1, x_2, y_1^\nu, y_2^\nu, y_3^\nu, y_4^\nu \in \mathbb{Z}_+
\end{aligned}
$$

Here, the random vector $\xi \in \mathbb{R}^n$ is given by the scenarios $\xi^1, \ldots, \xi^N$, all with equal probability $1/N$. The realizations of (ξ_1^ν, ξ_2^ν) and (ξ_3^ν, ξ_4^ν) are given by uniform grids (of differing granularity) in the squares $[300, 500] \times [300, 500]$ and $[0, 2000] \times [0, 2000]$, respectively. Timings are given in CPU seconds on Apple Silicon M1 chip.

It's easy to find that we have a feasible solution $x_1 = x_2 = y_1^\nu = y_2^\nu = 0, y_3^\nu = \xi_1^\nu$, and $y_4^\nu = \xi_2^\nu, \nu = 1, \ldots N$, which can be augmented to optimality.

First we change the program to the form below

$$
\begin{aligned}
\min \left\{ \frac{1}{N}\sum_{\nu=1}^{N} 35x_1 + 40x_2 + 16y_1^\nu + 19y_2^\nu + 47y_3^\nu + 54y_4^\nu \right\} \\
\text{s.t. } & x_1 + y_1^\nu + y_3^\nu - u_1^\nu = \xi_1^\nu \\
& x_2 + y_2^\nu + y_4^\nu - u_2^\nu = \xi_2^\nu \\
& 2y_1^\nu + y_2^\nu + u_3^\nu = \xi_3^\nu \\
& y_1^\nu + 2y_2^\nu + u_4^\nu = \xi_4^\nu, \\
& x_1, x_2, y_1^\nu, y_2^\nu, y_3^\nu, y_4^\nu, u_1^\nu, u_2^\nu, u_3^\nu, u_4^\nu \in \mathbb{Z}_+
\end{aligned}
$$

i.e.

$$
\min_{x \in X \subseteq \mathbb{R}^n} \left\{ f(x) := \frac{1}{N}\sum_{\nu=1}^{N} F\left(x, \xi_\nu\right) \right\},
$$

where

$$
\begin{aligned}
F\left(x_1, x_2, \xi_\nu\right) = \min \left(35x_1 + 40x_2 + 16y_1^\nu + 19y_2^\nu + 47y_3^\nu + 54y_4^\nu\right) \\
\text{s.t. } & x_1 + y_1^\nu + y_3^\nu - u_1^\nu = \xi_1^\nu \\
& x_2 + y_2^\nu + y_4^\nu - u_2^\nu = \xi_2^\nu \\
& 2y_1^\nu + y_2^\nu + u_3^\nu = \xi_3^\nu \\
& y_1^\nu + 2y_2^\nu + u_4^\nu = \xi_4^\nu, \\
& x_1, x_2, y_1^\nu, y_2^\nu, y_3^\nu, y_4^\nu, u_1^\nu, u_2^\nu, u_3^\nu, u_4^\nu \in \mathbb{Z}_+
\end{aligned}
$$

, which in matrix form denotes

$$
\begin{aligned}
F\left(x, \xi_\nu\right) = \min \, & c \cdot (x, y, u) \\
\text{s.t. } & A \begin{pmatrix} x \\ y^\nu \\ u^\nu \end{pmatrix} = \xi \\
& x_1, x_2, y_1^\nu, y_2^\nu, y_3^\nu, y_4^\nu, u_1^\nu, u_2^\nu, u_3^\nu, u_4^\nu \in \mathbb{Z}_+
\end{aligned}
$$

where $x = (x_1, x_2)^T$, $y^\nu = (y_1^\nu, y_2^\nu, y_3^\nu, y_4^\nu)^T$, $u^\nu = (u_1^\nu, u_2^\nu, u_3^\nu, u_4^\nu)^T$, $\xi^\nu = (\xi_1^\nu, \xi_2^\nu, \xi_3^\nu, \xi_4^\nu)^T$.

6.1 Algorithm 4.3

By importing the 'time' package in Python, we test the time of the program.

First, it took 131.44028520584106 seconds to compute the Graver Basis of A, which is a required ingredient for the programs corresponding to all scenarios.

Then we give the realization of $(\xi_1^\nu, \xi_2^\nu, \xi_3^\nu, \xi_4^\nu)$ by uniform grids (of differing granularity) in the squares $[300, 1300] \times [300, 1300] \times [200, 12000] \times [200, 12000]$ every three integers. For computing all 333 scenarios' optimum, i.e. to compute the opportunity-cost matrix, the time for the whole programs is 55.01750707626343 seconds.

6.2 Algorithm 3.1

First, it took 0.04420804977416992 seconds to compute the generating set of the toric ideal of A, which can be a required ingredient for the Gröbner Basis corresponding to all scenarios.

Then we give the realization of $(\xi_1^\nu, \xi_2^\nu, \xi_3^\nu, \xi_4^\nu)$ by uniform grids (of differing granularity) in the squares $[300, 1300] \times [300, 1300] \times [200, 12000] \times [200, 12000]$ every three integers. For computing all 333 scenarios' optimal points, i.e. to compute the opportunity-cost matrix, we apply the bunchberger algorithm first to calculate the Gröbner Basis of each scenario and then find the optimum each. The time for the whole programs is 9.76793909072876 seconds.

6.3 Remark

Algorithm 4.3 has the advantage that it calculate the Graver Basis once and for all. However, computational cost might boost quickly when the matrix A is not of small size. The new algorithm 3.1 fixes this by only computing the generating set of the generating set of the toric ideal of A, which can be very helpful to calculate the Gröbner Basis corresponding to all scenarios. After calculating the generating set, we can get Gröbner Basis of each scenario without any efforts. So the time of computing the opportunity-cost matrix is greatly reduced.

Algorithm 4.4 is an algorithm to calculate the Graver Basis of the whole program in special cases where $\mathrm{Ker}_{\mathbb{R}}(A) = \{0\}$. It might be costly for the same reason as algorithm 4.3. However, still it greatly reduces the computation.

7 Further study

1. Find a quicker algorithm combined with Graver Basis and Branch and Bound technique. (That is to combine algebraic technique and purely combinatorial technique, since purely algebraic relation is exhausted from my perspective.)

8 Codes

8.1 Algorithm 4.3

```
from sympy import *
import matplotlib.pyplot as plt
import numpy as np
from typing import Callable
import itertools
import random
import pyomo

# toric ideal of A
def toric_ideal(A):
    # Define symbolic variables ys for each row (index 0 in Python)
    sym_str_y = 'y:' + str(A.shape[0])
    ys = symbols(sym_str_y)

    # Define symbolic variables xs for each column (index 0 in Python)
    sym_str_x = 'x:' + str(A.shape[1])
    xs = symbols(sym_str_x)

    def to_polynomial(coef,vars):
        '''
        Function to define a single column of the coefficient as a polynomial
        '''
        res1 = 1
        res2 = 1
        for i in range(len(coef)):
            if coef[i] >= 0:
                res1 = res1*vars[i]**coef[i]
            else:
                res2 = res2*vars[i]**(-coef[i])
        res = res1 - res2
        return res

    def polynomial_ideal(A):
        '''
        Function to define a the polynomial ideal of a matrix A according to Conti and Traverso
        '''
        IA = A.col_insert(0, eye(A.shape[0]))
        # Find nullspace (kernel) of A
        ker = IA.nullspace()

        # Normalize elements of kernel to be integers
        ker_len = len(ker)
        for i in range(ker_len):
            rationalvector = True
            while rationalvector:
                factor = 1
                for j in ker[i]:
                    if j%1 != 0:
                        factor = min(factor,j%1)
                if factor == 1:
                    rationalvector = False
                else:
```

```
                    ker[i]=ker[i] / factor

        vars = ys + xs

        gen = []
        for k in ker:
            gen.append(to_polynomial(k,vars))


        return(gen, vars)

    IA, vars = polynomial_ideal(A)
    tor = groebner(IA, vars, order='lex')

    toric = []

    for i in tor:
        i = Poly(i)
        i_str = str(i.gens)
        #print(i_str)
        if not 'y' in i_str:
            toric.append(i)

    return toric, xs, ys

# Graver Basis of A
def GraverBasis(A):

    def Alaw(A):
        # n : column dimension r : row dimension
        A = Matrix(A)
        r = A.shape[0]
        n = A.shape[1]
        Id = np.concatenate((np.identity(n),np.identity(n)),axis = 1)
        Alaw = np.concatenate((A, np.zeros((r, n))),axis = 1)
        Alaw = np.concatenate((Alaw, Id),axis = 0)

        Afin = Alaw.astype(int)
        Afin = Matrix(Afin)
        return Afin, n

    def monomial(p):
        return [prod(x**k for x, k in zip(p.gens, mon)) for mon in p.monoms()]

    def to_T(toric):
        toric_fin=[]
        for g in toric:
            for k in range(n,2*n):
                g = g.subs({(xs[k],1)})
            toric_fin.append(g)

        toric_len = len(toric)

        vp = [0]*n
        vm = [0]*n
        T = []
```

```
        for k in range(0,toric_len):
            for i in range(0,n):
                p = monomial(Poly(toric_fin[k]))[0]
                m = monomial(Poly(toric_fin[k]))[1]
                vp[i] = degree(p,xs[i])
                vm[i] = degree(m,xs[i])
            v = np.array(vp) - np.array(vm)
            v = v.astype(int)
            T.append(v)
        return T

    Afin, n = Alaw(A)
    toric, xs, ys= toric_ideal(Afin)

    T = to_T(toric)
    return T

# augmentation algorithm
def augmentation(z_feas,c,T):
    # z_feaas: feasible point ; c: cost; T: universal test set
    exist_aug = True
    i = 0
    while exist_aug:
        exist_aug = False
        for t in T:
            if np.dot(c, t, out=None)>0 and np. all((z_feas-t>=0)):
                z_feas = z_feas-t
                i = 1+i
                #print('Iteration step', i,': vector', z_feas)
                exist_aug = True
            if np.dot(c, t, out=None)<0 and np. all((z_feas+t>=0)):
                z_feas = z_feas+t
                i = i+1
                #print('Iteration step', i,': vector', z_feas)
                exist_aug = True
    #print('Achieve an optimum!')
    return z_feas
```

8.2 Algorithm 4.4

```
def SpecGr_Com(Gr_1,m,N):
    SpecGr = []
    for k in range(1,N+1):
        for l in Gr_1:
            l_list = list(l)
            temp = [0]*m+[0]*(N-1)+l_list+[0]*(N-k)
            temp = np.array(l_list)
            SpecGr.append(temp)
    return SpecGr
```

8.3 Algorithm 3.1

```
def to_polynomial(coef,vars):
  '''
  Function to define a single row of the coefficient as a polynomial
  '''
```

```
  res1 = 1
  res2 = 1
  for i in range(len(coef)):
  if coef[i] > 0:
  res1 = res1*vars[i]**coef[i]
  elif coef[i] < 0:
  res2 = res2*vars[i]**(-coef[i])
  res = res1 - res2
  return res

def monomial(p):
    return [prod(x**k for x, k in zip(p.gens, mon)) for mon in p.monoms()]

def new_toric(A,c):
    # column/row dimension of a matrix
    r = A.shape[0]
    n = A.shape[1]
    A = np.array(A)

    # Define symbolic variables xs for each column (index 0 in Python)
    sym_str_x = 'x:' + str(A.shape[1])
    xs = symbols(sym_str_x)

    # Calculate a basis for the kernel
    def Ker(A):
        A = Matrix(A)
        B = np.array(Matrix(A).nullspace()).transpose()
        C = Matrix(B[0]).transpose()
        return C

    # find an equivalent basis with all base vectors lying in the same orthant
    def same_orthant(A):
        # compute the highest value in abstract value in a numpy natrix
        def abs_max(A):
            A = np.array(A)
            return abs(max(A.min(), A.max(), key=abs))

        for i in range(1,r):
            A[i,:] = A[i,:]+A[i-1,:]*(1+abs_max(A[i,:]))
        A_fin = np.array(A)
        A_fin = A_fin.astype(int)
        A_fin = Matrix(A_fin)
        return A_fin

    # compute J and prepare the function phi & phi_inv
    def nega_index(A):
        J = [ A[0,k]<0 for k in range(0,A.shape[1])]
        return J

    def reverse_sign(A):
        n = A.shape[1]
        A = np.array(A)
        for i in range(0,n):
            if nega_index(A)[i]:
                A[:,i] = - A[:,i]
        A_fin = A.astype(int)
        A_fin = Matrix(A_fin)
```

```
    return A_fin

def phi(A):
    G_J=[]
    for k in range(0,A.shape[0]):
        G_J.append(to_polynomial(A[k,:],xs))
    return G_J

def phi_inv(G_J):
    G_J_len = len(G_J)

    temp = []
    vp = [0]*n
    vm = [0]*n
    for k in range(0,G_J_len):
        for i in range(0,n):
            p = monomial(Poly(G_J[k]))[0]
            m = monomial(Poly(G_J[k]))[1]
            vp[i] = degree(p,xs[i])
            vm[i] = degree(m,xs[i])
        v = np.array(vp) - np.array(vm)
        v = v.astype(int)
        temp.append(v)
        temp_fin = Matrix(temp)
    return temp_fin, temp


def TJ_operation(G_J,J):
    for j in J:
        if j:
            # change position of xs[j]
            xs_list = list(xs)
            index = J.index(j)
            J[index] = False
            temp = xs_list[index]
            xs_list.remove(temp)
            xs_list.insert(0,temp)

            G_J = list(groebner(G_J, xs_list , order='lex'))

            A3, _ = phi_inv(G_J)
            A3[:,index] = - A3[:,index]
            G_J = phi(A3)

    return G_J


# MAIN
# generating set for the corresponding matrix kernel
A1 = Ker(A)

# matrix pre-operation
A2 = same_orthant(A1)
J = nega_index(A2)
A3 = reverse_sign(A2)
```

```
# groebner basis T_J operation
G_J = phi(A3)
G_zero = TJ_operation(G_J,J)

# define new order in Sympy
class compositeOrder(polys.orderings.MonomialOrder):
    """Composite-Lexicographic order of monomials. """
    alias = 'clex'
    is_global = True

    def __call__(self, monomial):
        return (np.dot(c,np.array(monomial)),monomial)
polys.orderings.__all__.append('clex')
clex = compositeOrder()
polys.orderings._monomial_key['clex'] = compositeOrder()

# apply groebner basis algorithm
G_fin = groebner(G_zero,xs,order='clex')
_ , G_fin = phi_inv(list(G_fin))

return G_fin
```

References

[1] Keutchayan, Julien, et al. “Problem-Driven Scenario Clustering in Stochastic Optimization.” ArXiv.org, 22 June 2021, https://arxiv.org/abs/2106.11717.

[2] Yuchen Ge. “Algebra for Machine Learning and Stochastic Programming.”August, https://gycdwwd.github.io/Writing/Algsto.pdf.

[3] Hemmecke, R., and R. Schultz. “Decomposition of Test Sets in Stochastic Integer Programming.” Mathematical Programming, vol. 94, no. 2-3, 2003, pp. 323–341., doi: https://doi.org/10.1007/s10107-002-0322-1.