Acta Univ. Sapientiae, Informatica, **16, 1** (2024) 20–38

DOI: 10.47745/ausi-2024-0002

A Memory-efficient Algorithm for Conservative Cuts in Disjoint Bilinear Programming

Xiaosong DING
International Business School
Beijing Foreign Studies University
Beijing, P.R.China, 100089
dingxiaosong@bfsu.edu.cn
0000-0003-3376-5774

Chao LIU
International Business School
Beijing Foreign Studies University
Beijing, P.R.China, 100089
202220316001@bfsu.edu.cn

Jun MA*
International Business School
Beijing Foreign Studies University
Beijing, P.R.China, 100089
*Corresponding author
202220316002@bfsu.edu.cn

Xi CHEN
International Business School
Beijing Foreign Studies University
Beijing, P.R.China, 100089
chenxi0109@bfsu.edu.cn
0000-0002-5803-8840

Qing SUN
International Business School
Beijing Foreign Studies University
Beijing, P.R.China, 100089
sy2009050823@163.com

Abstract. This paper presents a memory-efficient approach to the combinatorial challenges associated with the process in search of conservative cuts for degeneracy removal in disjoint bilinear programming. Drawing inspiration from Pascal's Triangle, the new randomized approach surpasses its predecessor from the perspective of memory, and thereby enables the possibility of its application to disjoint bilinear models with high degrees of degeneracy in imprecise decision analysis on a PC. Numerical reports are provided to validate its computational performance.

Key words and phrases: Conservative cuts, degeneracy, disjoint bilinear programming

1 Introduction

A typical disjoint bilinear programming (DBLP) intends to minimize a non-convex quadratic function over disjoint constraint sets defined by two bounded and non-empty polytopes. Mathematically, it can be formulated as below,

$$\begin{aligned} \min\ & f(\boldsymbol{x},\boldsymbol{y}) = \boldsymbol{c}^t\boldsymbol{x} + \boldsymbol{d}^t\boldsymbol{y} + \boldsymbol{x}^t\boldsymbol{Q}\boldsymbol{y}, \\ \text{s.t. } & \boldsymbol{x} \in \boldsymbol{X} = \{\boldsymbol{x} \in \mathbb{R}^{n_1} : \boldsymbol{A}\boldsymbol{x} \le \boldsymbol{a},\ \boldsymbol{x} \ge 0\}, \quad \boldsymbol{A} \in \mathbb{R}^{m_1 \times n_1},\ \boldsymbol{a} \in \mathbb{R}^{m_1}, \\ & \boldsymbol{y} \in \boldsymbol{Y} = \{\boldsymbol{y} \in \mathbb{R}^{n_2} : \boldsymbol{B}\boldsymbol{y} \le \boldsymbol{b},\ \boldsymbol{y} \ge 0\}, \quad \boldsymbol{B} \in \mathbb{R}^{m_2 \times n_2},\ \boldsymbol{b} \in \mathbb{R}^{m_2}. \end{aligned} \tag{1}$$

As a subset of bilinear programming, DBLP is a mathematical optimization framework that has gained significant attention due to its extensive applications in various fields including game theory, facility location, numerical linear algebra and stochastic processes; see [1]. More recent applications also cover DBLP related issues in supply chain management [2], [3], chemical engineering [4], two-dimensional packing [5], Markov decision process [6], [7], operations research [8], [9], etc. This paper focuses on the field of imprecise decision analysis, in which a typical decision model intends to solve many disjoint bilinear programs; see for example [10]–[15]. Each program in general possesses no more than 100 dimensions, but may encounter various degrees of degeneracy ranging from 1 to 5.

The unique challenge in solving DBLP lies in its bilinear objective function and disjoint constraints. Based on the structural properties, various solution techniques have been developed, among which, two major deterministic approaches are cutting plane methods and branch and bound methods.

In cutting plane methods, great effort has been devoted to the establishment of deep cuts like concavity cuts [16], polar cuts [17], decomposition cuts [18], etc. Nevertheless, the computational issue of degeneracy arising at a local solution can be frequently confronted in real-world applications. The development of an effective cut at a degenerate vertex has been long-standing with few computational results [17], [19]–[21]. Recently, the concept of conservative cuts was first introduced in [22]. From theoretical and computational viewpoints, a conservative cut neither sacrifices the local optimum by pivoting to a non-degenerate neighboring vertex [19], nor imposes too much computational load by generating a disjunctive cut [19], [23]. Accompanying the concept, a distance-following algorithm was proposed in search of a conservative cut, in which considerable computational effort is spent in the projection operation for each qualified adjacent vertex with respect to an established hyperplane. Additionally, the investigations of both a candidate's neighborhood and the distance between the degenerate vertex and an established hyperplane appear time-consuming. In [21], several heuristic algorithms aiming to further improve the computational performance for the location of a conservative hyperplane

were proposed, among which, **Algorithm 1** and **5** (will be referred to as **Random** throughout the rest of this paper; see **Appendix**), appeared quite promising in terms of computing time. By utilizing the property of non-uniqueness of a conservative hyperplane, **Random** is able to solve the bilinear models with low degrees of degeneracy arising in imprecise decision analysis, but may suffer from memory problems when the degree of degeneracy rises.

Since confronting heavy storage load of memory can severely restrict its application to a cutting plane method on PCs, we try to improve **Random** from the perspective of memory by utilizing Pascal's Triangle. With a static Pascal's Triangle table, we develop the inverse function of $nchoosek\_enum(n, k, i)$ to derive the serial number of a combination. It spares the need to save all combinations during the search in order to keep away from the memory problem. The improvement comes at a cost of computing time, and well illustrates the algorithmic space-time continuum in data structure.

In what follows, **Section 2** briefly describes the issue of degeneracy arising in the local optimization phase of a cutting plane method for DBLP. **Section 3** develops the memory-efficient algorithm for the location a conservative hyperplane. **Section 4** discusses the generation of required test instances, following which **Section 5** reports their computational performance based on the programs in imprecise decision analysis. **Section 6** concludes our paper.

2 Degeneracy

The essential solution property of DBLP exploited in the local optimization phase of almost all cutting plane methods is that even though $f(\boldsymbol{x}, \boldsymbol{y})$ is not quasi-concave, the global optimizer, $(\boldsymbol{x}^*, \boldsymbol{y}^*)$, is attained at a vertex of $\boldsymbol{X} \times \boldsymbol{Y}$, which means that $\boldsymbol{x}$ and $\boldsymbol{y}$ are vertices of $\boldsymbol{X}$ and $\boldsymbol{Y}$, respectively [24].

To facilitate our presentation, denote by $\boldsymbol{X}^i$ the original feasible region $\boldsymbol{X}$ when $i = 0$, or its subset obtained after i cuts have been introduced.

Definition 2.1. *A local minimizer of $g(\cdot)$ over $\boldsymbol{X}^i$ is a vertex, $\boldsymbol{x}_{\ell m}$, such that $g(\boldsymbol{x}_{\ell m}) \leq g(\boldsymbol{x})$ for each $\boldsymbol{x} \in \mathcal{B}_\delta(\boldsymbol{x}_{\ell m}) \cap \boldsymbol{X}^i$, where $\mathcal{B}_\delta(\boldsymbol{x}_{\ell m})$ is a δ-neighborhood around $\boldsymbol{x}_{\ell m}$ in $\boldsymbol{X}^i$, and $g(\boldsymbol{x}_{\ell m})$ is the corresponding local minimum.*

Definition 2.2. *A local star minimizer of $g(\cdot)$ over $\boldsymbol{X}^i$ is a vertex, $\boldsymbol{x}_{\ell sm}$, such that $g(\boldsymbol{x}_{\ell sm}) \leq g(\boldsymbol{x})$ for each $\boldsymbol{x} \in \mathcal{N}(\boldsymbol{x}_{\ell sm})$, where $\mathcal{N}(\boldsymbol{x}_{\ell sm})$ denotes the vertices adjacent to $\boldsymbol{x}_{\ell sm}$ in $\boldsymbol{X}^i$, and $g(\boldsymbol{x}_{\ell sm})$ is the corresponding local star minimum.*

Since $f(\boldsymbol{x}, \boldsymbol{y})$ is not quasi-concave, a local star minimum is not necessarily a local minimum, and thus the development of a cut from a local star minimizer cannot take

effect as usual for those with quasi-concave objective functions. Moreover, for DBLP (1), cuts involving variables associated with both $\boldsymbol{X}^i$ and $\boldsymbol{Y}$ may destroy their special structure, and thereby fail the existing efficient algorithms to solve sub-problems. As a result, to develop a cut that involves only the $\boldsymbol{x}$-variables and yet is convergent from a local minimizer, a concept more than **Definition 2.1, 2.2** is necessary [25].

Definition 2.3. *A vertex $(\overline{\boldsymbol{x}}^i, \overline{\boldsymbol{y}})$ in DBLP is a Pseudo-Global Minimizer (PGM) if $f(\overline{\boldsymbol{x}}^i, \overline{\boldsymbol{y}}) \leq f(\boldsymbol{x}, \boldsymbol{y})$ for each $\boldsymbol{x} \in \mathcal{B}_\delta(\overline{\boldsymbol{x}}^i) \cap \boldsymbol{X}^i$ and for each $\boldsymbol{y} \in \boldsymbol{Y}$.*

For DBLP (1), a vertex is adjacent to $(\overline{\boldsymbol{x}}^i, \overline{\boldsymbol{y}})$ if and only if it is either of the form $(\boldsymbol{x}^k, \overline{\boldsymbol{y}})$ or $(\overline{\boldsymbol{x}}^i, \boldsymbol{y}^k)$ where $\boldsymbol{x}^k \in \mathcal{N}_{\boldsymbol{X}^i}(\overline{\boldsymbol{x}}^i)$ and $\boldsymbol{y}^k \in \mathcal{N}_{\boldsymbol{Y}}(\overline{\boldsymbol{y}})$. For a PGM, further improvement may be achieved by an idea analogous to that suggested by **Definition 2.2**, i.e., we can examine those vertices adjacent to $\overline{\boldsymbol{x}}^i$ for a better solution. A so derived PGM can have the advantages from both a local minimum and a local star minimum. **Algorithm 1**, originated from [24] to identify a PGM, $(\overline{\boldsymbol{x}}^i, \overline{\boldsymbol{y}})$, is currently acting as a building block in the local optimization phase of a cutting plane method.

Algorithm 1: Augmented Mountain Climbing Method

Input: $\boldsymbol{Q}, \boldsymbol{c}, \boldsymbol{d}, \boldsymbol{X}^i, \boldsymbol{Y}, \widetilde{\boldsymbol{y}} \in \boldsymbol{Y}$.
Output: $(\overline{\boldsymbol{x}}^i, \overline{\boldsymbol{y}})$.
1 **repeat**
2 $\quad \widetilde{\boldsymbol{x}} = \arg\min_{\boldsymbol{x} \in \boldsymbol{X}^i} f(\boldsymbol{x}, \widetilde{\boldsymbol{y}})$; $\quad \widetilde{\boldsymbol{y}} = \arg\min_{\boldsymbol{y} \in \boldsymbol{Y}} f(\widetilde{\boldsymbol{x}}, \boldsymbol{y})$;
3 **until** $\widetilde{\boldsymbol{x}}$ *converges*;
4 construct $\mathcal{N}_{\boldsymbol{X}^i}(\widetilde{\boldsymbol{x}})$;
5 **if** $\exists \check{\boldsymbol{x}} \in \mathcal{N}_{\boldsymbol{X}^i}(\widetilde{\boldsymbol{x}})$ *such that* $f(\check{\boldsymbol{x}}, \boldsymbol{y}^*) = \min_{\boldsymbol{y} \in \boldsymbol{Y}} f(\check{\boldsymbol{x}}, \boldsymbol{y}) < f(\widetilde{\boldsymbol{x}}, \widetilde{\boldsymbol{y}})$ **then**
6 $\quad$ go to **line 2** with $\widetilde{\boldsymbol{y}} = \boldsymbol{y}^*$;
7 **terminate** with $(\overline{\boldsymbol{x}}^i, \overline{\boldsymbol{y}}) = (\widetilde{\boldsymbol{x}}, \widetilde{\boldsymbol{y}})$ as a PGM.

In **Algorithm 1**, it turns out that $\overline{\boldsymbol{x}}^i$ in a PGM, $(\overline{\boldsymbol{x}}^i, \overline{\boldsymbol{y}})$, derived in **line 7**, can be degenerate. This will result in the inevitable computational difficulty in the establishment of a valid cut in the global optimization phase since we will have more than n_1 cutting points along the edges emanating from $\overline{\boldsymbol{x}}^i$, not to mention its effectiveness and efficiency.

Degeneracy can be further classified into weak degeneracy and strong degeneracy. In a two-dimensional (2D) program, as has always been done in the literature, it is only possible to introduce weak degeneracy by bringing in some redundant constraints; see **Example 1**.

Example 1. As shown in the left sub-figure of **Figure 1**, suppose that the reduced feasible region is defined by $\mathbb{P} \cap \mathbb{Q}$ as

$$\mathbb{P} = \left\{(x_1, x_2)^t : 0 \le x_1, x_2 \le 1\right\},$$
$$\mathbb{Q} = \left\{(x_1, x_2)^t : 4x_1 + x_2 \ge 1,\ \frac{4}{3}x_1 + x_2 \le 1\right\},$$

where $\mathbb{P}$ and $\mathbb{Q}$ can be regarded as the constraints to define the original feasible region and introduced cuts, respectively.

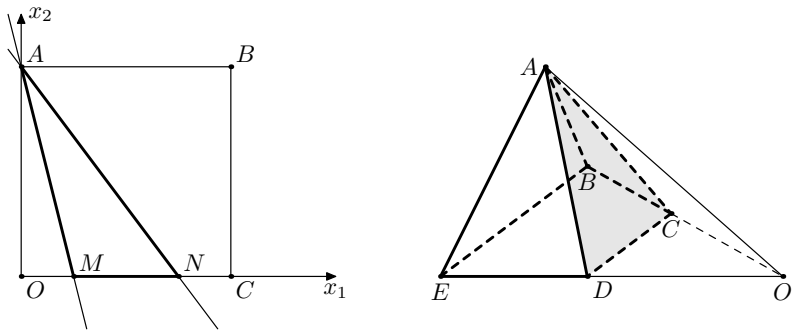

Figure 1: Weak Degeneracy versus Strong Degeneracy.

The feasible region in **Example 1** is bounded by three bold lines. Apparently, the degeneracy at vertex $\boldsymbol{A}$ can be resolved by removing two redundant linear constraints, $x_1 \ge 0$ and $x_2 \le 1$.

Intuitively, weak degeneracy occurs simply because of redundant constraints, and can be avoided by carrying out some pre-processing procedure [26], [27]. For any 2D program, barring the extreme case where the feasible region consists of a single point, only weak degeneracy exists, i.e., more than two linear constraints intersect at a single point.

In **Figure 1**, a three-dimensional (3D) instance in the right sub-figure illustrates strong degeneracy, in which $\boldsymbol{ACD}$ acts as an introduced cut to remove a portion of the feasible region $\boldsymbol{OACD}$. As a result, there are four edges emanating from $\boldsymbol{A}$ in the reduced feasible region. Having derived their respective maximal step-sizes, we can hardly expect four cutting points to be coplanar, thus leading to the computational difficulty in the generation of a valid cut from $\boldsymbol{A}$ in $\mathbb{R}^3$. Note that the removal of any constraint cannot take effect in the resolution of strong degeneracy because it changes the feasible region.

3 Conservative Cuts

3.1 Basic Knowledge

According to our experience, strong degeneracy is frequently confronted in DBLP (1) with multi-global minima. With one of the global minima derived as the current best objective value, a cut can at most reach another global minimum and may induce degeneracy therein. Consequently, it is essential to develop an efficient and effective approach for establishing a valid cut from a degenerate vertex. A promising technique specially designed for this purpose is conservative cuts.

In what follows, we will omit the superscript i in $\overline{\boldsymbol{x}}^i$ and $\boldsymbol{X}^i$ given a clear context. That is, in the i^{th} iteration, we simply take $\overline{\boldsymbol{x}}$ as $\overline{\boldsymbol{x}}^i$ in a PGM, $(\overline{\boldsymbol{x}}^i, \overline{\boldsymbol{y}})$, provided by **Algorithm 1**, and $\boldsymbol{X}$ as $\boldsymbol{X}^i$, the reduced feasible region. Besides, the procedure in search of a conservative hyperplane is formulated in $\mathbb{R}^n$ rather than $\mathbb{R}^{n_1}$ as in (1).

Geometrically, for a polytope defined by $\boldsymbol{X}$ in $\mathbb{R}^n$, a degenerate vertex $\overline{\boldsymbol{x}}$ has more than n incident edges. During the development of a cut, say, a polar cut, given the appropriate step-sizes along their positive or negative extensions, the probability that all cutting points are coplanar is fairly low. We thus have more than n points in $\mathbb{R}^n$ to establish a cut that should not exclude any potential optimal solution [22].

Denote by $\Delta = \mathcal{N}_{\boldsymbol{X}_0}(\overline{\boldsymbol{x}}) = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n, \boldsymbol{x}_{n+1}, \boldsymbol{x}_{n+2}, \ldots, \boldsymbol{x}_{n+\sigma}\}$ the set of all vertices adjacent to $\overline{\boldsymbol{x}}$, by σ ($\sigma \geq 1$) the degree of degeneracy, by Ω ($\Omega \subset \Delta$) the set containing n vertices selected from Δ to establish a hyperplane Π, and by $\overline{\Omega}$ the set of adjacent vertices that lie to the same side of Π as $\overline{\boldsymbol{x}}$ does.

Definition 3.1. *In $\mathbb{R}^n$, given a degenerate vertex, $\overline{\boldsymbol{x}}$, of a polytope, a conservative hyperplane, Π^*, is defined as the hyperplane generated by n vertices neighboring to $\overline{\boldsymbol{x}}$ such that $\overline{\Omega} = \emptyset$.*

Definition 3.2. *In $\mathbb{R}^n$, at a degenerate vertex, $\overline{\boldsymbol{x}}$, of a polytope, a conservative cut used to cut off $\overline{\boldsymbol{x}}$ is the inequality generated by a conservative hyperplane.*

3.2 Pascal's Triangle

Random, by utilizing implicit enumeration, acts as a promising procedure in search of a conservative cut. Although **Random** selects a candidate randomly in exchange for one of the vertices establishing the current hyperplane, it remains inevitable to keep a record on those already visited combinations. Otherwise, we may sacrifice the computing time for visiting the same combination and even step into infinite loops. In MATLAB, the function $nchoosek(\boldsymbol{V}, K)$ suits well for this purpose. It returns a matrix containing all possible combinations of the elements of vector $\boldsymbol{V}$ taken K at a time.

Suppose we confine an instance to the dimension of 125, which is sufficiently high for programs in imprecise decision analysis. To figure out a conservative cut, a PC equipped with 8G memory will throw an error such as "Requested $234531275x5$ (8.7GB) array exceeds maximum array size preference (8.0GB)" when **Random** carries out $nchoosek(\boldsymbol{V}, 5)$ where $\boldsymbol{V} = [1, 2, \ldots, 125]^t$. Though we can partially resolve this issue by configuring "MATLAB array size limit" in "Workspace", the generation process will run extremely slow and cause MATLAB to become unresponsive. This is reasonable because $nchoosek(\boldsymbol{V}, 5)$ intends to enumerate a total of 234531275 combinations. They will then serve as the indices to locate corresponding adjacent vertices in Δ to establish $\boldsymbol{\Omega}$. The number of combinations of $nchoosek(\boldsymbol{V}, K)$ increases particularly fast as K rises. We can hardly expect each PC is equipped with 8G memory so that the aforementioned memory problem can severely hinder **Random**'s application.

To circumvent this issue, we introduce a critical technique that is able to take effect throughout the implementation of our memory-efficient approach. To avoid building the full combination array in memory like what $nchoosek(\boldsymbol{V}, K)$ does, we take advantage of the function $nchoosek\_enum(n, k, i)$ with an enumerating selection of the i^{th} combination.

Consider an example for all six combinations of C_4^2, i.e., $(1, 1, 2)$, $(2, 1, 3)$, $(3, 1, 4)$, $(4, 2, 3)$, $(5, 2, 4)$, and $(6, 3, 4)$, with the first entry as the corresponding serial number. The 3^{rd} combination of C_4^2 derived by $nchoosek\_enum(4, 2, 3)$ is $(1, 4)$, the 5^{th} combination by $nchoosek\_enum(4, 2, 5)$ is $(2, 4)$, etc. By referring to the Pascal's Triangle table, we need to develop its inverse, $inverse\_nchoosek\_enum(n, k, \boldsymbol{c})$, such that, given an appropriate combination $\boldsymbol{c}$, we can derive its serial number. By "appropriate", we mean herein that all elements in $\boldsymbol{c}$ are organized in accordance with their natural order. Still with the previous example, the output for $inverse\_nchoosek\_enum(4, 2, [3, 4])$ is 6. The rationale can be found in **Example 1** with detailed explanations.

Example 1. Here comes an example with a detailed workflow for illustration. We list only the necessary rows of $Pascal(9)$ provided by MATLAB for our purpose; see **Table 1**.

We take the following procedures to figure out the serial number corresponding to the combination, $\boldsymbol{c} = [3, 5, 6, 8, 9]$, among all combinations of choosing 5 elements out of $\boldsymbol{V} = [1, 2, \ldots, 9]^t$. For the first element 3, two preceding elements are 1 and 2. First, for 1, there are eight elements, i.e., $2, 3, \ldots, 9$, left for the rest four positions, and therefore it possesses $C_8^4 = 70$ combinations in total. Once the four elements are selected, their order is fixed in consistence with the natural order. Next, for 2, there are seven elements, i.e., $3, 4, \ldots, 9$, left for the rest four positions, and therefore it

Table 1: Pascal(9)

1	1	1	1	1	1	1	1	1
1	2	③	4	5	6	7	8	9
1	3	6	⑩	15	21	28	36	45
1	4	10	20	㉟	56	84	120	165
1	5	15	35	⑦⓪	126	210	330	495

possesses $C_7^4 = 35$ combinations in total. When it comes to 5, we need to count the number of combinations starting with $[3, 4, \ldots]$. By an analogous logic, it is $C_5^3 = 10$ in total. Finally, we come to 6, which has C_3^2 combinations with $[8, 9]$ as the last one. As a result, the output of $inverse\_nchoosek\_enum(9, 5, [3, 5, 6, 8, 9])$ is $C_8^4 + C_7^4 + C_5^3 + C_3^2 = 70 + 35 + 10 + 3 = 118$, as indicated by the circled numbers.

Note that it is unnecessary to handle $[5, 6]$ and $[8, 9]$ in $[3, 5, 6, 8, 9]$ because they are two pairs of consecutive natural numbers.

Example 1 illustrates that by choosing appropriate numbers from a static Pascal's Triangle table, we can readily derive the serial number of an appropriate combination. Generally speaking, it is unnecessary to save a complete Pascal's Triangle table in memory because on the one hand, we could rarely expect such an ugly degenerate problem with $\sigma > 5$. On the other hand, since a Pascal's Triangle table is symmetric, the storage is so cheap that it can serve as a very efficient and effective means of deriving the serial number of some combination of choosing K out of N elements.

3.3 Implementation

With $inverse\_nchoosek\_enum()$, denote by $\#$ the serial number of an appropriate combination of choosing K out of N elements, by $\boldsymbol{v}$ the normal vector of Π^*, by $\boldsymbol{A}$ and $\boldsymbol{b}$ the parameters in the constraint set, $\boldsymbol{Ax} \leq \boldsymbol{b}$, by Q a structure to save the serial numbers of those Ωs whose candidates for exchange have already been exhausted; and by $\overline{Q}$ a structure array to save those Ωs with the same $|\overline{\Omega}|$, where $|\cdot|$ is the cardinality of a set. The so constructed structure array $\overline{\Omega}$ indexed by $|\overline{\Omega}|$ is used to accelerate the search among candidates. Moreover, there exists another structure, $abnormal$, which saves those Ωs constituting (close to) singular matrices. Such Ωs are considered inappropriate to establish hyperplanes.

Algorithm 2 realizes the memory-efficient randomized algorithm with details. Notice that it is the essential property of non-uniqueness of a conservative hyper-

plane that enables **Random** to surpass its other counterparts in [21], [22] and be competitive with **Algorithm 2** in terms of different σs.

Algorithm 2: Conservative Hyperplane (Improved Randomized)

Input: $\Delta, \overline{\boldsymbol{x}}$
Output: $\boldsymbol{v}, \Omega$

1 set $\boldsymbol{v} = \emptyset$;
2 **while** $\boldsymbol{v} = \emptyset$ **do**
3 randomly extract # from $\overline{Q}_\ell$ where $\ell = \arg\min_{\ell'}\{\ell' | \overline{Q}_{\ell'} \neq \emptyset\}$;
4 **if** $\ell = -1$ **then**
5 randomly collect n vertices from Δ into Ω to set up a Π;
6 **if** $|\overline{\Omega}| = 0$ **then return** $\boldsymbol{v}$ of Π^* and Ω;
7 **if** $\# \notin \overline{Q} \cup Q \cup abnormal$ **then**
8 insert # of Ω, and all $\boldsymbol{x}_k \in \overline{\Omega}$ into $\overline{Q}_{|\overline{\Omega}|}$;
9 **else**
10 recover Ω corresponding to the extracted #;
11 randomly select $\boldsymbol{x}_j \in \overline{\Omega}$, and set $\overline{\Omega} = \overline{\Omega} \backslash \{\boldsymbol{x}_j\}$;
12 **if** $|\overline{\Omega}| = 0$ **then** move # of Ω from $\overline{Q}_\ell$ to Q;
13 **foreach** $\boldsymbol{x}_k \in \Omega$ **do**
14 exchange $\boldsymbol{x}_j$ with $\boldsymbol{x}_k$ to set up Ω' and Π';
15 **if** $|\overline{\Omega'}| = 0$ **then return** $\boldsymbol{v}$ of Π^* and Ω';
16 **if** $\# \notin \overline{Q} \cup Q \cup abnormal$ **then**
17 insert # of Ω', and all $\boldsymbol{x}_k \in \overline{\Omega'}$ into $\overline{Q}_{|\overline{\Omega'}|}$;

In **Algorithm 2**, in order to validate the availability of one Ω, it is necessary to check Q, $\overline{Q}$, and $abnormal$ first; see **line 7, 16**. We can save in Q, $\overline{Q}$ and $abnormal$ only the serial number associated with each Ω rather than specific elements. This can greatly alleviate the usage of memory since it is unnecessary to visit all combinations in search of a conservative hyperplane. Otherwise, it would become quite unclear on the memory size we need to allocate in advance, or an arbitrary allocation may simply lead to "out of memory"; see **line 3**, **7**, **12** and **16**. Additionally, the utilization of actual combinations will compel the program to compare a sorted candidate with each entry, e.g., $ismember()$ in MATLAB. The comparison of vectors may also slow down the computations.

Several other aspects need to be further clarified. Firstly, a normal vector, $\boldsymbol{v}$, exists

provided that Ω is well-defined to establish a hyperplane, i.e., Ω is not in *abnormal*, otherwise $\boldsymbol{v}$ is kept empty. As a result, the only stopping criterion is either $|\overline{\Omega}| = 0$ or $|\overline{\Omega'}| = 0$ regardless of $\boldsymbol{v}$; see **line 6, 15**. One exception is that, in **line 12**, $|\overline{\Omega}| = 0$ indicates that all candidates for exchange regarding some Ω have been exhausted so that we should move it from $\overline{Q}$ to Q. Secondly, although it is suggested that Ω or $\overline{\Omega}$ in **Algorithm 2** contain qualified adjacent vertices, we actually take advantage of their row indices in Δ to indicate their positions and then extract them. By doing so, we intend to relieve the storage load of memory. Thirdly, in **line 4**, $\ell = -1$ means that no Ω exists for search, which may arise provided that all available Ωs have been exhausted or when initializing the entire algorithm. Finally, in **line 7, 16**, each operation inserts into $\overline{Q}$ one serial number and appends to it a set containing all qualified vertices. The algorithm extracts these candidates one by one for exchange until the set becomes empty (**line 11**, and **line 13** through **17**).

4 Test Instances

Algorithm 3 is used to generate a test instance with $n + \sigma$ vertices adjacent to a σ-degenerate vertex $\overline{\boldsymbol{x}}$ in $\mathbb{R}^n$.

Algorithm 3: Test Instances

Input: n, σ
Output: Δ

1 generate a polytope with $n+1$ points $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n, \boldsymbol{x}_{n+1}\}$ in $\mathbb{R}^n$;
2 fix one point, say, $\boldsymbol{x}_{n+1}$ as $\overline{\boldsymbol{x}}$, and collect the remaining into Δ;
3 **while** $|\Delta| \leq n + \sigma$ **do**
4 select $\boldsymbol{x}_i \in \Delta$;
5 **foreach** $\boldsymbol{x}_j$, $j \neq i$, *adjacent to* $\boldsymbol{x}_i$ **do**
6 generate a new point $\boldsymbol{x}_{|\Delta|+j}$ and set $\Delta = \Delta \cup \{\boldsymbol{x}_{|\Delta|+j}\}$

$$\boldsymbol{x}_{|\Delta|+j} = \sum_{k \neq j} \alpha_{jk} \boldsymbol{x}_k,\ \alpha_{jk} \in (0,1),\ k \neq j,\ \sum_{k \neq j} \alpha_{jk} = 1;$$

7 set $\Delta = \Delta \backslash \{\boldsymbol{x}_i\}$, and re-order so that $\Delta = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_{|\Delta|}\}$;
8 randomly select $n + \sigma$ points from Δ so that $\Delta = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_{n+\sigma}\}$;
9 **foreach** $\boldsymbol{x}_i \in \Delta$ **do** randomly extend or shorten $\overline{\boldsymbol{x}}\boldsymbol{x}_i$;

In **Algorithm 3**, between **line 3** and **7**, the while loop will not stop until the number of elements in Δ exceeds $n + \sigma$, the required number of adjacent vertices.

This provides us with great flexibility in the generation of a degenerate instance, despite the cost of some excessive computational effort. In **line 6**, the operation is in fact the convex combination of $\boldsymbol{x}_i$ with all the remaining $\boldsymbol{x}_k$s except $\boldsymbol{x}_j$ for each $j \neq i$. Therefore, we generate $n-1$ new points in total. Together with $\overline{\boldsymbol{x}}$, a hyperplane can be generated to remove $\boldsymbol{x}_i$, and the number of vertices adjacent to $\overline{\boldsymbol{x}}$ becomes $|\Delta| + n - 2$. In **line 9**, by randomly extending or shortening $\overline{\boldsymbol{x}}\boldsymbol{x}_i$, there may still exist more than n points on some hyperplane. However, a test instance is considered inappropriate only if all adjacent vertices lie on the same hyperplane. By carrying out the critical operation in **line 9**, this could seldom happen.

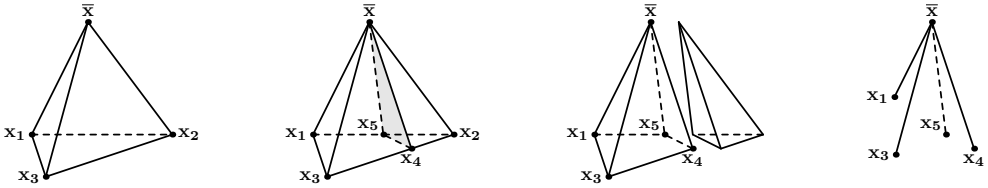

Figure 2: Illustration of **Algorithm 3**.

Algorithm 3 can be illustrated by a 3D example in **Figure 2**, where $n = 3$ and $\sigma = 1$. By **Figure 2**, each construction of a cut to remove some $\boldsymbol{x}_i$ will generate one more point adjacent to $\overline{\boldsymbol{x}}$. As a result, we are able to generate any number of points adjacent to $\overline{\boldsymbol{x}}$, regardless of how high the degree of degeneracy is.

5 Numerical Results

In order to evaluate the performance of two randomized algorithms, especially **Algorithm 2**, we take advantage of **Algorithm 3** to generate the required test instances and carry out the experiments on a PC equipped with Intel(R) Core(TM) i5-6267U CPU @ 2.90GHz and 4G memory. We deliberately impose a limit of 60 seconds over the total computing time for each instance, which appears reasonable for an interactive decision analysis software package. For each combination of dimension (n) and degree of degeneracy (σ), we generate 24 test instances and average their corresponding results with respect to total computing time, the number of performing $nchoosek\_enum(n, k, i)$, etc. In the following, we intend to evaluate the performance of two algorithms for test instances with low σ and n, high σ and low n, and low σ and high n, respectively. The exception is for test instances with high σ and n due to the memory issue raised by $nchoosek(n, k)$. With the current environment, we can only try to increase either n or σ, but not both, as will be demonstrated by the experiments.

5.1 Low Degree of Degeneracy and Dimension

Figure 3 illustrates the performance for test instances with low σ ($\sigma = 1, 2, 3$) and n ($n = 30, 35, \ldots, 100$). Three sub-figures on the first row of **Figure 3** illustrate the performance of two algorithms regarding $\sigma = 1, 2, 3$, respectively. It can be observed that **Random** runs faster than **Algorithm 2** most of the time. Nevertheless, the gaps in terms of computing time are rather small, say, within only around 1.5 seconds. Two sub-figures on the second row of **Figure 3** illustrate their individual performance with respect to $\sigma = 1, 2, 3$. For both algorithms, the computing time increases exponentially as n rises given $\sigma = 2, 3$, whereas the performance of **Random** keeps relatively steady when $\sigma = 1$, as apposed to that of **Algorithm 2**.

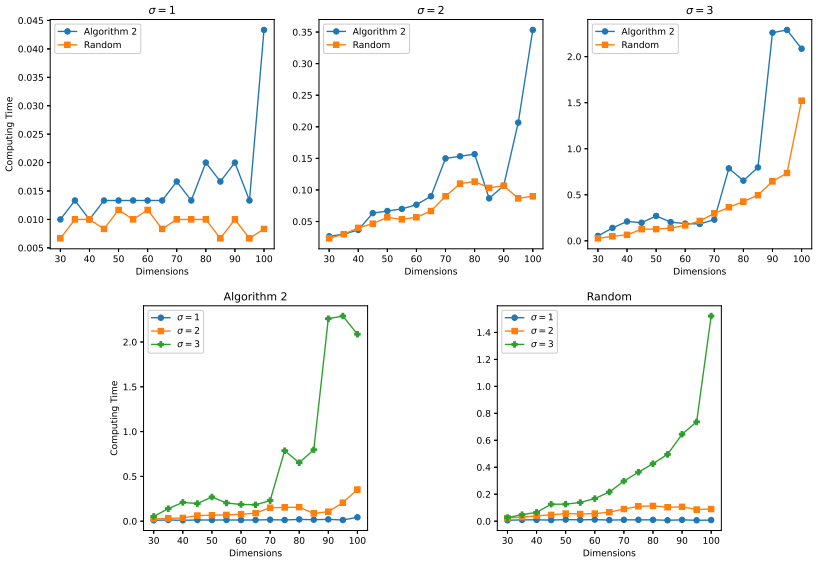

Figure 3: Performance of **Algorithm 2** and **Random** (Low σ and n).

The fact that **Random** runs faster than **Algorithm 2** in the current setting appears reasonable due to the sufficient memory. **Random** can take advantage of $nchoosek(\boldsymbol{V}, K)$, which generates all possible combinations and save them in memory. However, as n or σ increases, the memory problem will emerge.

5.2 High Degree of Degeneracy and Low Dimension

Figure 4 illustrates their results for instances with high σ ($\sigma = 4, 5$) when n goes from 30 to 100. Two sub-figures on the first row of **Figure 4** demonstrate their performance. Provided $\sigma = 4$, **Random** can run up to around $n = 100$ before its computing time exceeds the pre-specified time limit, or the test instance incurs "out of memory", whereas provided $\sigma = 5$, similar situations take place around $n = 75$. To some extent, this demonstrates that each increase in σ can dramatically impact the computational performance of **Random**. Meanwhile, the computing time of **Algorithm 2** is acceptable. By two sub-figures on the first row of **Figure 4**, the performance of **Algorithm 2** dominates that of **Random** for $\sigma = 4, 5$. Two sub-figures on the second row of **Figure 4** illustrate the number of performing nchoosek_enum(n,k,i) and the number of solving linear equations, respectively, regarding **Algorithm 2** when $\sigma = 4, 5$. In the bottom-left sub-figure, the numbers of performing nchoosek_enum(n,k,i) are competitive when $n \leq 70$. However, when $n \geq 75$, a clear increase in the number for $\sigma = 5$ can be observed, as compared with that for $\sigma = 4$. Besides, a higher σ, in general, corresponds to a larger number of solving linear equations; see the bottom-right sub-figure of **Figure 4**.

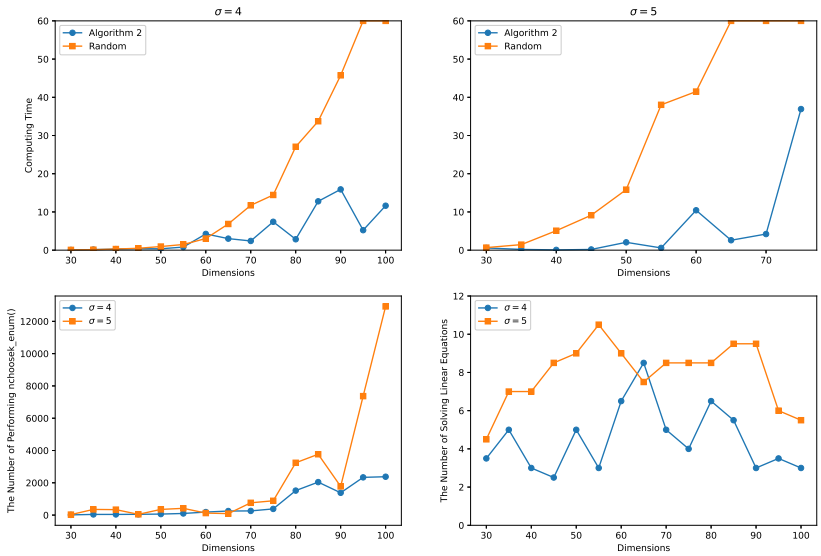

Figure 4: Performance of **Algorithm 2** and **Random** (High σ, Low n).

Most computing time of **Algorithm 2** is spent in $nchoosek\_enum(n,k,i)$ to generate serial numbers in exchange for memory. By contrast, solving linear equations only accounts for a very small portion of total computing time. As σ rises, the computing time rises sharply. Nonetheless, the impact from the rise in n on the total computing time seems much less when σ is not that high. Additionally, the higher the σ, the more number of $nchoosek\_enum(n,k,i)$ **Algorithm 2** performs. The trend becomes much clearer as n rises. This is reasonable since the number of candidates in search of a conservative hyperplane increases exponentially. By contrast, $inverse\_nchoosek\_enum(n,k,\boldsymbol{c})$ costs almost nothing since the Pascal's Triangle table is static.

5.3 Low Degree of Degeneracy and High Dimension

Figure 5 illustrates their performance for instances with low σ ($\sigma = 1,2,3$) and high n ($n = 100,105,\ldots,250$). For the purpose of comparison, it is impossible to perform **Random** with respect to these dimensions when $\sigma = 4,5$ due to the pre-specified time limit or "out of memory" issue; see also the previous illustration when $n \leq 100$. It can be observed that for $\sigma = 1$, their performance overlaps most of the time. Nevertheless, for $\sigma = 2,3$, **Algorithm 2** outperforms **Random** across all most dimensions, which demonstrates its ability and qualification in handling problems with higher dimensions. Note also that the gaps in computing time for $\sigma = 1,2$ are relatively small, showing two algorithms are very competitive.

6 Conclusions

This paper investigates an improved randomized algorithm, i.e., **Algorithm 2**, in addressing the memory challenge associated with the previously developed searching process, **Random**, for a conservative cut. By leveraging the inherent structure of Pascal's Triangle, the memory-efficient **Algorithm 2** well illustrates the algorithmic space-time continuum in data structure.

Computational experiments demonstrate that **Algorithm 2** is particularly beneficial to programs characterized by high n or σ, whereas **Random** is more suitable for programs with low σ. However, once the memory problem appears, **Random** will become less preferred so that **Algorithm 2** should come into play. By refining **Random**, we anticipate its application to a cutting plane method in imprecise decision analysis where DBLP plays a pivotal role.

What should be noted is that the Matlab environment utilized herein is an interpreter compiler, which slows down the test significantly unless a special-purpose command line was used in the compilation. To accelerate, an even better approach

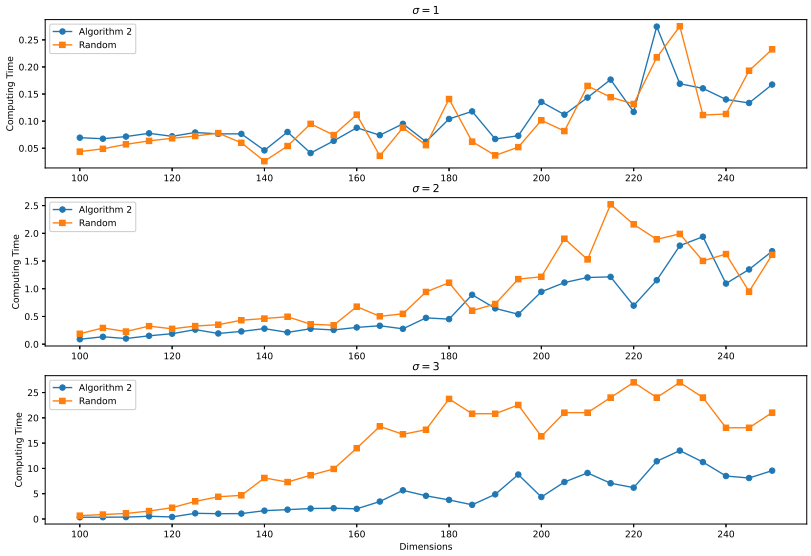

Figure 5: Performance of **Algorithm 2** and **Random** (Low σ and High n).

is to develop the entire cutting plane method using C/C++.

Author Contributions: Conceptualization, Xiaosong DING and Xi CHEN; Methodology, Xiaosong Ding and Chao LIU; Formal Analysis, Jun MA and Xi CHEN; Software, Xiaosong Ding; Writing – original draft, Xiaosong Ding; Writing – review & editing, Xi CHEN and Qing SUN. All authors have read and agreed to the manuscript's published version.

Funding: This research was partially funded by the Fundamental Research Funds for the Central Universities (2022TD001, 2023JJ002) and Young Faculty Research Fund of BFSU (2019-ZYQNJS-002).

Data Availability: This work did not generate new data.

Acknowledgments: We would like to thank two anonymous reviewers for their insightful remarks.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix

In $\mathbb{R}^n$, suppose that in a polytope, a degenerate vertex, $\overline{\boldsymbol{x}}$, has been located with $n+m$ adjacent vertices, $\Delta = \{B_1, B_2, \ldots, B_n, B_{n+1}, B_{n+2}, \ldots, B_{n+m}\}$. By randomly selecting n points, we can obtain $\mathcal{R}_\Theta$ to establish a hyperplane Ω and $\mathcal{S}_{\hat{\Theta}}$ with $|\mathcal{S}_{\hat{\Theta}}| = k$ where $|\cdot|$ is the number of elements in a set. For each possible k, we set up a queue, $\overline{Q}_k$, to save all $\mathcal{R}_\Theta$s with $|\mathcal{S}_{\hat{\Theta}}| = k$ that have been found but not yet utilized as the starting set of points for the branching process. Similarly, we establish another queue, Q_k, to save all $\mathcal{R}_\Theta$s with $|\mathcal{S}_{\hat{\Theta}}| = k$ that have already been utilized for the branching process.

Algorithm 4: General Algorithm

Input: $\Delta, \overline{\boldsymbol{x}}$
Output: $\mathcal{R}_\Theta$

randomly choose n points from Δ as $\mathcal{R}_\Theta$ and establish Ω with $\mathcal{R}_\Theta$;
update with $\ell = |\mathcal{S}_{\hat{\Theta}}| = k$, and save $\mathcal{R}_\Theta$ in $\overline{Q}_\ell$;
while $\ell \neq 0$ **do**
    extract one $\mathcal{R}_\Theta$ from $\overline{Q}_\ell$;
    update with $\overline{Q}_\ell = \overline{Q}_\ell / \{\mathcal{R}_\Theta\}$, $Q_\ell = Q_\ell \cup \{\mathcal{R}_\Theta\}$;
    ℓ = **Algorithm 5**($\mathcal{R}_\Theta$);
return $\mathcal{R}_\Theta$.

Algorithm 5: Branching Algorithm (Randomized)

Input: $\mathcal{R}_\Theta$
Output: ℓ

randomly choose one R_i in $\mathcal{R}_\Theta$ and one S_j in $\mathcal{S}_{\hat{\Theta}}$;
set $\mathcal{R}_{\Theta'} = \{\mathcal{R}_\Theta / \{R_i\}\} \cup \{S_j\}$;
if $\mathcal{R}_{\Theta'} \notin Q_{k'} \cup \overline{Q}_{k'}$, $\left(|\mathcal{S}_{\hat{\Theta}'}| = k'\right)$ **then** save $\mathcal{R}_{\Theta'}$ in $\overline{Q}_{k'}$, and set $\ell = k'$;
else $\ell = \arg\min_m \{m | \overline{Q}_m \neq \emptyset\}$;
return ℓ.

References

[1] H. Konno, "Bilinear programming: Part II. Application of bilinear programming," Tech. Rept. No. 71-10, Department of Operations Research, Stanford University, Stanford, Calif., 1971 (⇒ 21).

[2] S. Rebennack, A. Nahapetyan, and P. Pardalos, "Bilinear modeling solution approach for fixed charge network flow problems," *Optimization Letters*, vol. 3, no. 3, pp. 347–355, 2009 (⇒ 21).

[3] A. Nahapetyan, "Bilinear programming: Applications in the supply chain management," in *Encyclopedia of Optimization*, Springer, 2009, pp. 282–288 (⇒ 21).

[4] D. Wicaksono and I. Karimi, "Piecewise MILP under- and overestimators for global optimization of bilinear programs," *AIChE Journal*, vol. 54, no. 4, pp. 991–1008, 2008 (⇒ 21).

[5] A. Caprara and M. Monaci, "Bidimensional packing by bilinear programming," *Mathematical programming*, vol. 118, no. 1, pp. 75–108, 2009 (⇒ 21).

[6] M. Petrik and S. Zilberstein, "Robust approximate bilinear programming for value function approximation," *The Journal of Machine Learning Research*, vol. 12, pp. 3027–3063, 2011 (⇒ 21).

[7] M. Valdebenito, C. Pérez, H. Jensen, and M. Beer, "Approximate fuzzy analysis of linear structural systems applying intervening variables," *Computers & Structures*, vol. 162, no. 1, pp. 116–129, 2016 (⇒ 21).

[8] A. Nahapetyan and P. Pardalos, "A bilinear relaxation based algorithm for concave piecewise linear network flow problems," *Journal of Industrial and Management Optimization*, vol. 3, no. 1, p. 71, 2007 (⇒ 21).

[9] A. Nahapetyan and P. Pardalos, "A bilinear reduction based algorithm for solving capacitated multi-item dynamic pricing problems," *Computers & Operations Research*, vol. 35, no. 5, pp. 1601–1612, 2008 (⇒ 21).

[10] M. Danielson, "Generalized evaluation in decision analysis," *European Journal of Operational Research*, vol. 162, no. 2, pp. 442–449, 2005 (⇒ 21).

[11] M. Danielson and L. Ekenberg, "A framework for analysing decisions under risk," *European Journal of Operational Research*, vol. 104, no. 3, pp. 474–484, 1998 (⇒ 21).

[12] M. Danielson and L. Ekenberg, "Computing upper and lower bounds in interval decision trees," *European Journal of Operational Research*, vol. 181, no. 2, pp. 808–816, 2007 (⇒ 21).

[13] M. Danielson and L. Ekenberg, *Real-Life Decision-Making*, 1st ed. CRC Press, 2023, ISBN: 9781003406709 (⇒ 21).

[14] A. Sage and C. White, "Ariadne: A knowledge-based interactive system for planning and decision support," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 14, no. 1, pp. 35–47, 1984 (⇒ 21).

[15] A. Salo and R. Hämäläinen, "Preference programming through approximate ratio comparisons," *European Journal of Operational Research*, vol. 82, no. 3, pp. 458–475, 1995 (⇒ 21).

[16] H. Tuy, "Concave programming under linear constraints," *Soviet Maththematics*, vol. 5, pp. 1437–1440, 1964 (⇒ 21).

[17] H. Sherali and C. Shetty, "A finitely convergent algorithm for bilinear programming problems using polar cuts and disjunctive face cuts," *Mathematical Programming*, vol. 19, no. 1, pp. 14–31, 1980 (⇒ 21).

[18] M. Porembski, "How to extend the concept of convexity cuts to derive deeper cutting planes," *Journal of Global Optimization*, vol. 15, no. 4, pp. 371–404, 1999 (⇒ 21).

[19] S. Alarie, C. Audet, B. Jaumard, and G. Savard, "Concavity cuts for disjoint bilinear programming," *Mathematical Programming*, vol. 90, no. 2, pp. 373–398, 2001 (⇒ 21).

[20] M. Porembski, "On the hierarchy of γ-valid cuts in global optimization," *Naval Research Logistics*, vol. 55, no. 1, pp. 1–15, 2008 (⇒ 21).

[21] X. Chen, J. Zhang, X. Ding, T. Yang, and J. Qian, "Location of a conservative hyperplane for cutting plane methods in disjoint bilinear programming," *Optimization Letters*, vol. 13, no. 7, pp. 1677–1692, 2019 (⇒ 21, 28).

[22] J. Zhang, X. Chen, and X. Ding, "Degeneracy removal in cutting plane methods for disjoint bilinear programming," *Optimization Letters*, vol. 11, no. 3, pp. 483–495, 2017 (⇒ 21, 25, 28).

[23] C. Audet, P. Hansen, B. Jaumard, and G. Savard, "A symmetrical linear maxmin approach to disjoint bilinear programming," *Mathematical Programming*, vol. 85, no. 3, pp. 573–592, 1999 (⇒ 21).

[24] H. Konno, "A cutting plane algorithm for solving bilinear programs," *Mathematical Programming*, vol. 11, no. 1, pp. 14–27, 1976 (⇒ 22, 23).

[25] H. Vaish and C. Shetty, "A cutting plane algorithm for the bilinear programming problem," *Naval Research Logistics*, vol. 24, no. 1, pp. 83–94, 1977 (⇒ 23).

[26] T. Gal, "A method for determining redundant constraints," in *Redundancy in Mathematical Programming*, Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 36–52, ISBN: 978-3-642-45535-3 (⇒ 24).

[27] T. Gal, "Weakly redundant constraints and their impact on postoptimal analyses in LP," *European Journal of Operational Research*, vol. 60, no. 3, pp. 315–326, 1992 (⇒ 24).

Received: 26.01.2024; *Revised: 13.05.2024*; *Accepted: 10.06.2024*