POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Automatic Matrix Decomposition for Reducing Performance Variability in Mixed-Integer Programming Solvers

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Luca Pedranzini**

Student ID: 251453
Advisor: Prof. Pietro Belotti
Academic Year: 2024-2025

Abstract

When it comes to modelling optimization problems for real-world scenarios coming from sectors like finance, production planning or network design it is necessary to have integer variables. This makes Mixed-Integer Programming essential when we want to solve optimization problems in these fields. Extensive research has been conducted into how these problems should be formulated and various improvements to the algorithms used to solve them have been made during the last 50 years.
Nevertheless, there is a feature which has been discovered more recently that has not been addressed much in research papers so far. We are talking about performance variability, which stands for different runtimes measured when solving shuffled but mathematically equivalent versions of the same problem. This thesis aims to build a framework to experiment with various optimization problem instances, investigate the causes of performance variability more thoroughly, and try to mitigate its effect through the reorderings of rows and columns of problem coefficient matrices. The framework allowed us to simulate adversarial behaviour where, given a problem instance, we generated different equivalent forms according to well-defined transformations. Once done with that, we experimented with different methods to rearrange rows and columns of the different forms in order to reach a unique form which ideally would lead to more stable solving times.
While doing that, we not only measured runtime variability but we also tried to define other metrics based on matrices structures that were correlated with solve time variability, and gave us more information about what are the features of a problem that influence solver behaviour. Doing that, we tried to get a better understanding of the transformations that occur to coefficients matrices when processed by state-of-the-art solver software.

Keywords: Mixed-Integer Programming, solver performance, variability, automatic decomposition

Abstract in lingua italiana

Quando si tratta di modellare problemi di ottimizzazione che emergono da scenari del mondo reale provenienti da settori come la finanza, la pianificazione della produzione o la progettazione di reti, è necessario utilizzare variabili intere. Questo rende la Programmazione Mista Intera (Mixed-Integer Programming) indispensabile quando si vogliono risolvere problemi di ottimizzazione in questi ambiti. Negli ultimi 50 anni sono state condotte ricerche approfondite su come questi problemi dovrebbero essere formulati e sono stati apportati numerosi miglioramenti agli algoritmi utilizzati per risolverli. Tuttavia, esiste una caratteristica emersa più recentemente che finora non è stata affrontata in modo significativo nella letteratura scientifica. Stiamo parlando della variabilità delle prestazioni, ovvero della differenza nei tempi di esecuzione misurati quando si risolvono formulazioni matematicamente equivalenti dello stesso problema ottenute riordinando le righe e le colonne. Questa tesi si propone di costruire un framework per sperimentare con varie istanze di problemi di ottimizzazione, indagare meglio le cause della variabilità delle prestazioni e cercare di mitigarne l'effetto attraverso il riordinamento di righe e colonne della matrice dei coefficienti del problema. Il framework ci ha permesso di simulare un comportamento avversario in cui, data un'istanza di problema, generiamo diverse forme equivalenti secondo trasformazioni ben definite. Una volta fatto ciò, sperimenteremo con diversi metodi per riordinare le righe e le colonne delle varie forme al fine di ottenere una forma unica che, idealmente, porterà a tempi di risoluzione più stabili. Nel fare ciò, non ci siamo limitati a misurare la variabilità del tempo di esecuzione, ma cercheremo anche di definire altre metriche basate sulla struttura delle matrici che siano correlate con la variabilità del tempo di risoluzione. Queste metriche ci forniranno maggiori informazioni sulle caratteristiche di un problema che influenzano il comportamento del risolutore, permettendoci così di comprendere meglio le trasformazioni subite dalle matrici dei coefficienti durante l'elaborazione da parte degli algoritmi più avanzati disponibili oggi.

Parole chiave: Mixed-Integer Programming, prestazioni dei solver, variabilità, decomposizione automatica

Contents

Introduction

When it comes to solving optimization problems, Linear Programming (LP) is a powerful technique used to model and solve problems where the objective function and constraints are linear. When these problems only involve continuous variables, they can be solved by applying algorithms such as the Simplex Method. They are widely applicable in different areas but mostly in logistics and production planning. However, many real-world problems have features that can only be modeled with discrete variables. That was the baseline that led to the development of Mixed-Integer Programming (MIP) as a general modeling framework that encompasses the discrete requirement while maintaining linearity in constraints and objective. A MIP problem consists of an objective function to be maximized or minimized, a set of linear constraints, and a mix of integer and continuous decision variables. Sometimes MIP problems may take a more specific form such as in the case when all variables are integers, or all variables are binary. MIP formulations have wider applications than LP, spanning from financial portfolio selection to network design. The presence of integer variables makes solving MIP problems significantly more complex than solving LPs. The most common exact solution methods to solve these problems include Branch-and-Bound (B&B), which uses a tree structure to explore feasible solutions and Branch-and-Cut (B&C), which integrates cutting-plane techniques to improve efficiency. Often heuristics are used to find good solutions when exact methods become computationally intractable, which can happen quite frequently given the NP-hard nature of MIP [5].

The most well-known commercial solvers are Gurobi [10], IBM CPLEX [12] and FICO Xpress [6], but there is also a vast scene of open source alternatives. In both cases, sophisticated algorithms are leveraged by solvers and hardware optimizations accelerate solving time for the largest and most complex problem instances. All these solvers are widely available to use with many common programming languages such as Python or Julia, enabling both researchers and professionals to apply them in their daily tasks. Despite that, MIP remains computationally challenging, with formulation pitfalls, scalability issues and numerical stability concerns affecting runtime performance.

1 | Problem Definition and Theoretical Foundations

1.1. Mathematical formulation

MIP problems are sometimes also called Mixed-Integer Linear Optimization (MILO) problems. These two terms identify the same subclass of optimization problems, with the second emphasizing more the linear nature of constraints and objective. They can be formally defined using two equivalent formulations:

1.1.1. Component-wise Notation

$$\begin{aligned} \min & \sum_{i \in N} c_i x_i \\ \text{s.t.} \quad & \sum_{i \in N} a_{ji} x_i \leq b_j, \quad \forall j = 1, 2, \ldots, m \\ & x_i \in \mathbb{Z}, \quad \forall i \in J \subseteq N \end{aligned}$$

where:

- $N = \{1, \ldots, n\}$ is the index set of all variables.
- $J \subseteq N$ is the subset of integer-constrained variables.
- c_i represents the cost coefficient for variable x_i.
- a_{ji} denotes the coefficient of variable x_i in constraint j.
- b_j is the right-hand side value of constraint j.
- The problem has n total variables, with $p = |J|$ being integer and $q = n - p$ being continuous.

1.1.2. Matrix Notation

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x_i \in \mathbb{Z}, \quad \forall i \in J \subseteq N \end{aligned}$$

where:

- $x \in \mathbb{R}^n$ is the decision variable vector.
- $c \in \mathbb{Q}^n$ is the cost coefficient vector.
- $A \in \mathbb{Q}^{m \times n}$ is the constraint coefficient matrix.
- $b \in \mathbb{Q}^m$ is the right-hand side constraint vector.
- $J \subseteq N$ defines the indices of integer-constrained variables.

These formulations are standard representations of Mixed-Integer Linear Programming (MIP) problems used in MILO solvers.

1.2. Data Collection

Turning a real-world problem into a Mixed-Integer Programming (MIP) model requires gathering a lot of data from different sources, which is often a time-consuming step that directly impacts its quality and effectiveness. Usually, the process for defining a problem involves multiple stakeholders, including domain experts, clients, operation researchers and data providers to ensure that all relevant aspects of the problem are captured accurately. When it comes to data, all goes down to extracting it from the sources available, including operational systems, databases, historical records and sensor data. Since real-world data is rarely clean and structured in an ideal format, significant effort is needed to preprocess and refine it before it can be integrated into a MIP model. Examples of this preprocessing are dealing with missing data and inconsistencies. In many cases, assumptions and approximations must be made due to the inherited uncertainty and variability in real-world datasets.

After all data has been gathered, it needs to be mapped into a mathematical formulation where models feature such as objective function, variables, constraints and bounds are defined and need to reflect the actual business logic that the model is trying to represent. Often, large-scale problems require aggregation or decomposition techniques to make data manageable while preserving the essential structure of the problem. The last step is

validation of the model, by making sure it represents the real-world scenario and adjusts it based on computational feasibility. Often in this phase, historical data can be used to back test and spot inconsistencies or errors in the formulation. Sometimes part of the problem requires data coming from real-time streams of data that can be obtained by integrating ERP systems into the optimization framework.

What is important to notice is that the exact same problem can be written in an infinite number of ways, leading to different model structures that have the same solution, meaning they are perfectly equivalent from the mathematical point of view.

1.3. Problem description: variability

This thesis addresses the phenomenon of variability, which can be frequently observed and is mainly caused by humans, for example, when it comes to deciding in which order to insert variables or constraints into the formulation of a new model. Variability can be defined as the dispersion [16] of a dataset, quantifying how much individual observations deviate from a central value such as the mean. We are going to assess variability of different metrics, even though what in literature has been mainly discussed so far is variability of solve time. In mathematical terms, the standard deviation (σ) serves as our primary measure of variability and it is computed as:

$$\sigma = \sqrt{\frac{1}{M-1}\sum_{j=1}^{M}(y_j - \bar{y})^2}$$

where y_j represents individual data points, $\bar{y}$ is the sample mean, and M is the number of observations. A higher standard deviation indicates greater variability among the values.

Variability [15] has been observed widely in industry and has been cited in many research papers as one of the factors to consider when experimenting with new solver features. The causes of variability must be searched among different reasons. One of them is that different researchers may express constraints in alternative yet equivalent ways, which depends on the person's sensibility, and it is reasonable to happen, knowing that each problem has many equivalent formulations. Also, some models may aggregate variables or constraints differently, which can influence how solvers perform decomposition. Another possibility is that the names and ordering of constraints and variables can vary, and coefficients can be scaled without altering the problem structure. For example, the following row inequality:

$$a^\top x \leq b$$

is equivalent to

$$\beta a^\top x \leq \beta b \quad \text{for any } \beta > 0.$$

Similarly, in the case of column scaling, consider a continuous variable x_1 appearing in a problem of the form:

$$x_1 \in \left\{ x \in \mathbb{R}^n : c^\top x \text{ minimized}, \, Ax \leq b \right\}.$$

This can be equivalently rewritten by introducing a scaled variable $y_1 = \beta x_1$, and updating the corresponding coefficient c_1 and column A_1 by a factor β, which results in an equivalent formulation with respect to optimal solutions. Lastly, as we cannot know in detail how the specific algorithm used by the solver works when decomposing a particular instance, considering a different order of variables and constraints may result in a different order of operations which implies a different solving time.

Researchers can agree on a unique method to formulate and aggregate constraints, which may lead to a more standard approach to the problem and eliminate variations generated by personal preference. On the other hand, permutation and scaling of variables and constraints are solely dependent on how the client provides the data, what data sources are being used, and with which priority we are gathering information from them.

What has been noticed about the impact variability has on solving time is that it is highly visible even for a small set of permuted or scaled variables. Being a phenomenon known for some years now, many experiments have been made that were not directly related to removing it but needed to take it into account. It has been observed that to fully manifest its impact during experiments with the most diffused commercial solvers, 4 to 5 variations of the same problem are often enough, as adding more of them does not increase variability significantly.

Permutation

Consider the following mixed-integer programming problem:

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x_i \in \mathbb{Z}, \quad \forall i \in J \subseteq N \end{aligned}$$

where $A \in \mathbb{R}^{m \times n}$ is the constraint matrix, $x \in \mathbb{R}^n$ is the vector of decision variables, $c \in \mathbb{R}^n$ is the objective coefficient vector, and $b \in \mathbb{R}^m$ is the right-hand-side vector. We can define a permutation of the problem in terms of permutation matrices.

A permutation matrix is a square binary matrix P that reorders elements of a vector or matrix when multiplied. Formally, a permutation matrix is obtained from the identity matrix I_n by permuting its rows. If π is a permutation of $\{1, \ldots, n\}$, the corresponding permutation matrix P satisfies:

$$Pe_i = e_{\pi(i)}$$

where e_i is the i-th standard basis vector.

In the context of problem permutations, we define two key permutation matrices:

- Variable (column) permutation matrix $P_{\text{col}} \in \mathbb{R}^{n \times n}$, which reorders the decision variables.
- Constraint (row) permutation matrix $P_{\text{row}} \in \mathbb{R}^{m \times m}$, which reorders the constraints.

Let $\tilde{x} = P_{\text{col}} x$ denote the permuted variable vector. Applying these permutations, the transformed version of the original problem becomes:

$$\begin{aligned} \min \quad & (P_{\text{col}}^T c)^T \tilde{x} \\ \text{s.t.} \quad & (P_{\text{row}} A P_{\text{col}}^T)\tilde{x} \leq P_{\text{row}} b \\ & \tilde{x}_j \in \mathbb{Z}, \quad \forall j \in J' := \{\pi(i) \mid i \in J\} \end{aligned}$$

The integrality constraints must also be updated consistently with the variable permutation. There are two equivalent ways to express this:

- In terms of the original variables x: $\quad x_i \in \mathbb{Z},\ \forall i \in J \subseteq N$.
- In terms of the permuted variables $\tilde{x} = P_{\text{col}} x$: $\quad \tilde{x}_{\pi(i)} = (P_{\text{col}} x)_{\pi(i)} \in \mathbb{Z},\ \forall i \in J \subseteq N$, where π is the permutation corresponding to P_{col}.

Equivalently, we can define the set of integer-constrained indices in the permuted variable space as $J' := \{\pi(i) \mid i \in J\}$, and write $\tilde{x}_j \in \mathbb{Z},\ \forall j \in J'$.

Since permutation matrices are orthogonal (i.e., $P^{-1} = P^T$), they preserve the fundamental properties of the problem. The feasible region remains the same, and the optimal solution, in transformed coordinates, corresponds directly to the original solution under the inverse permutation.

From a computational perspective, problem permutations can impact solver runtime due to differences in numerical stability, branching order in branch-and-bound methods, and presolve effectiveness. This makes permutations a useful tool for analyzing solver behavior and robustness in large-scale optimization.

Scaling

In a similar way we can define scaling, which this time affects the magnitude of coefficients inside the matrix and vector formulation of the problems.

Consider a mixed-integer linear optimization problem given by:

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x_i \in \mathbb{Z}, \quad \forall i \in J \subseteq N, \end{aligned}$$

where $A \in \mathbb{R}^{m \times n}$, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, and $N = \{1, \ldots, n\}$.

To scale this problem, we introduce two invertible diagonal matrices:

- Column scaling matrix: $D_{\text{col}} = \text{diag}(s_1, s_2, \ldots, s_n)$ with $s_j \neq 0$ for all $j \in N$.
- Row scaling matrix: $D_{\text{row}} = \text{diag}(r_1, r_2, \ldots, r_m)$ with $r_i > 0$ for all $i = 1, \ldots, m$.

Define a new variable $y \in \mathbb{R}^n$ by the relation:

$$y = D_{\text{col}} x \quad (\text{so that } x = D_{\text{col}}^{-1} y).$$

Then, the objective function becomes:

$$c^T x = c^T (D_{\text{col}}^{-1} y) = \left(D_{\text{col}}^{-T} c\right)^T y = \tilde{c}^T y,$$

and the constraints transform as:

$$Ax \leq b \quad \Leftrightarrow \quad A(D_{\text{col}}^{-1} y) \leq b \quad \Leftrightarrow \quad D_{\text{row}} A D_{\text{col}}^{-1} y \leq D_{\text{row}} b.$$

The scaled problem is therefore:

$$\begin{aligned} \min \quad & \tilde{c}^T y \\ \text{s.t.} \quad & \tilde{A} y \leq \tilde{b} \\ & y_j \in s_j \mathbb{Z}, \quad \forall j \in J, \end{aligned}$$

where:

$$\tilde{c} = D_{\text{col}}^{-T} c, \quad \tilde{A} = D_{\text{row}} A D_{\text{col}}^{-1}, \quad \tilde{b} = D_{\text{row}} b.$$

Since D_{row} and D_{col} are invertible, this transformation is reversible, preserving both feasibility and optimality.

Impact on solver performance

The problem transformations described above can have non-trivial effects on solver runtime and numerical performances. That is because even though the problem is equivalent and has an equivalent set of optima, the path a solver takes to reach the solution may vary especially when permutations are considered. The main impact of scaling is on numerical stability, since solvers typically rely on floating-point arithmetic and different coefficients lead to slightly different rounding that may sum up. Similarly, permutations can lead to small rounding errors that may accumulate. The phase when this impact can be observed is in the presolve steps. It happens quite often in state-of-the-art commercial solvers that before starting the actual problem solution, the solver attempts to simplify and change the order in which variables are considered later for solving. That is why both permutations and numerical instability can determine differences already in this phase. After that solvers usually implement heuristics to select the next variables to branch on. Every branching decision determines the path we are going to follow in a search tree which can lead to significant runtime differences. Of course, if the first branching decisions are different due to a permutation we will end up exploring very different search trees which will be very likely to lead to different runtimes.

Example: Variability in Runtime for `neos5`

We consider the `neos5` problem instance from the MIPLIB [19] as an example of runtime variability. This is a relatively small problem, with 63 variables and 63 constraints, that has a fast solve time, yet it still clearly demonstrates variability caused by different permutations. Figure 1.1 shows the original problem and two permutations: white squares correspond to null coefficients a_{ij}.

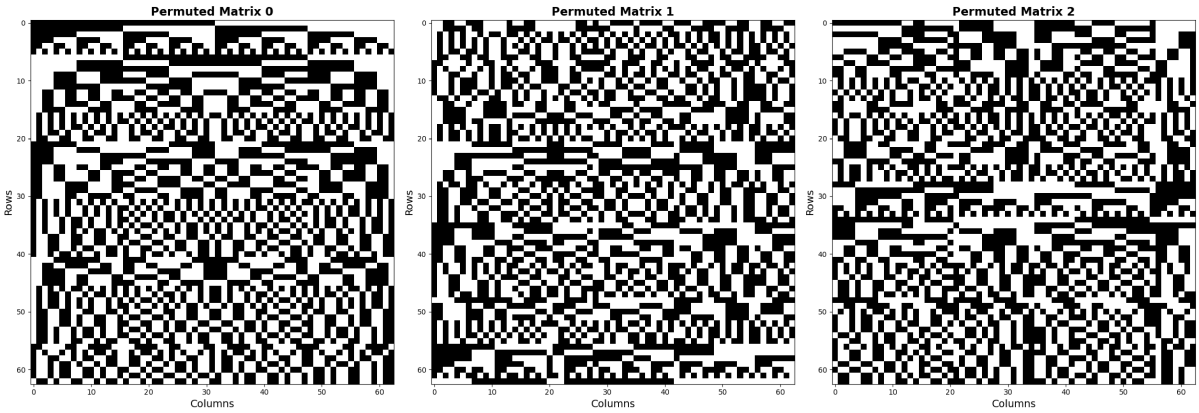

Figure 1.1: Original and two permutations of `neos5`

To evaluate the impact of permutations on solver performance, we compare the runtime (measured in seconds) for the original instance and the two permuted versions. The original runtime is 102.94 seconds, while the two permutations took 63.99 seconds and 98.16 seconds, respectively. These results indicate that permutations can significantly affect solver behavior: one permutation reduces runtime by approximately 38%, while the other is close to the original runtime.

To quantify this variability, we compute the mean and standard deviation of these runtimes. The mean runtime is

$$\mu = \frac{102.94 + 63.99 + 98.16}{3} \approx 88.36 \text{ seconds},$$

and the variance is calculated as

$$\sigma^2 = \frac{(102.94 - 88.36)^2 + (63.99 - 88.36)^2 + (98.16 - 88.36)^2}{3} \approx 300.88,$$

which gives a standard deviation (our chosen measure of variability) of

$$\sigma = \sqrt{300.88} \approx 17.34 \text{ seconds}.$$

This example highlights that permutations of the same instance can lead to substantial differences in runtime. Although the underlying mathematical problem is unchanged, solver components such as preprocessing, branching strategies, and heuristics may be influenced by the instance formulation, resulting in performance variability.

1.4. Computational complexity

Solver efficiency is obviously impacted by the computational complexity of the problem considered. As stated earlier MIP problems are inherently NP-hard, which means that the worst-case solution time exponentially scales with the input size. For this reason, solving these problems is a computationally challenging task even when using state-of-the-art solvers. Problem formulation has an impact on the complexity as it affects the tightness of the linear relaxations. A tighter formulation, which approximates more closely the integer feasible region when variables are relaxed, often leads to fewer branch-and-bound nodes, which results in an improved solver performance. On the other hand, looser formulations have the effect of increased computational efforts due to weaker bounds and extensive branching requirements. At this point, we can better understand how even small

variations of the problem can have quite a big impact, as even swapping two columns in the matrix representation of the problem might lead to different branching strategies or relaxations, that consequently lead to large deviations in performance across very close instances in terms of formulation.

Ideally, we should aim to find a reordering of the problem where the solver quickly discards infeasible or suboptimal branches instead of redundantly exploring a large portion of the search space. That would mean directing our efforts to find a unique reordering that also has better performances. Even though it may be an interesting objective for future research, the following chapters will only aim to find a reordering that is able to mitigate the variability phenomena, which is not necessarily the one that also minimizes solve time.

2 | Problem Setup and Evaluation Metrics

2.1. Towards Permutation-Invariant Representations of MIP Instances

As discussed previously, variability in MIP problems represents a challenge for solver efficiency estimation and stability, representing also an obstacle in the development of better solvers. That is because observing a particular problem being solved very fast by a new algorithm gives little to no information on how fast a different reordering of the same can be solved by the exact same new algorithm.

To address this issue, this thesis introduces a heuristic reordering algorithm that reduces the influence of arbitrary permutations and scaling, to achieve more stable solver performance and improve the effectiveness of modern branch-and-bound solvers. To build such a heuristic, we need to investigate the structural properties of MIP problems and come up with a set of rules that leverage this information, with the aim of reformulating the problem in a more standard way that is hopefully nearly unique for every variation of the same problem.

The main challenge of building something similar is that it is probably not universally applicable to all categories of instances. The reason is that some problems do not exhibit clear structural patterns that can be exploited by a reordering heuristic. On the other hand, some problems have very clear structures and can be recognized very precisely. That is why the final challenge is also to identify which categories of problems are more suitable to be considered when experimenting with the heuristic and which ones gain more benefits in terms of variability when we apply it.

2.2. Distance measure

While laying the foundations to experiment with different heuristics, one of the fundamental metrics we need to define is a measure of how different are two equivalent forms of the same problem. At this point, the focus will be solely on measuring different permutations of the problems. The primary reason is that the heuristics should be scaling invariant to be robust and the only way we could also keep scaling into account would be to, at some point, apply some kind of normalization steps, which may clash with the goal of maintaining a certain numerical stability.

Given two different reorderings of the same problem, obtained by applying two different permutation matrices of the form seen before, a measure of the difference between them is what will be called "permutation distance" from now on. Different options can be considered to formalize this measure, but the choice that seemed more appropriate is the Kendall Tau distance [13], a well-established metric for measuring the pairwise inversions between two matrix permutations. An inversion occurs when two elements appear in a different relative order between two permutations.

More formally, given a permutation ϕ_1 and a permutation ϕ_2 of n elements, the Kendall Tau distance counts the number of index pairs (i, j) such that $i < j$ and one of the following holds:

- $\phi_1(i) > \phi_1(j)$ but $\phi_2(i) < \phi_2(j)$, or
- $\phi_1(i) < \phi_1(j)$ but $\phi_2(i) > \phi_2(j)$.

When this distance is low, it indicates that the two permutations are closely related. On the other hand, a high distance means that the two permutations are significantly different. This measure has to be compared with solver efficiency, and what is expected is that low distances mean closer solving time, so less variability. In the context of the experiment that will be done to reduce variability, this distance will come in handy to assess the effectiveness of a heuristic, that will be naturally considered better whenever leads to closer permutation distances between the various reordering to which it is applied.

Here are two examples demonstrating Kendall Tau distance with visual representations of permutation matrices. These illustrate how different permutations affect distance and solver variability.

Example 1: Low Kendall Tau Distance

Consider two permutations ϕ_1 and ϕ_2 of $n = 5$:

- $\phi_1 = [1, 2, 3, 4, 5]$ (identity permutation)
- $\phi_2 = [1, 3, 2, 4, 5]$ (a single swap)

Permutation matrices are representations of permutations where each row corresponds to the original order, and each column shows the new position of each element.

Matrix for ϕ_1

$$P_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrix for ϕ_2

$$P_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The only inversion occurs between elements $(2, 3)$ since in ϕ_1, $2 < 3$, but in ϕ_2, $3 < 2$. Thus, the Kendall Tau distance is:

$$K(\phi_1, \phi_2) = 1$$

This low distance suggests that solver performance should be similar for both orders.

Example 2: High Kendall Tau Distance (Dissimilar Permutations)

Consider:

- $\phi_1 = [1, 2, 3, 4, 5]$
- $\phi_2 = [5, 4, 3, 2, 1]$ (reverse order)

Matrix for ϕ_2

$$P_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Computing Kendall Tau Distance Every pair (i, j) is inverted, leading to:

$$K(\phi_1, \phi_2) = \binom{5}{2} = 10$$

Since the distance is maximum, these two permutations are very different. This suggests that solver performance might vary significantly between these permutations.

2.3. Runtime measure

To have a clearer measurement of the effectiveness of the ordering algorithm, we will need to consider not only permutation distance but also solve time variability. That is because in literature it is one of the few metrics that has been used so far to measure variability, consequently the results of the experiment can only be compared and judged based on that. Nevertheless, the distance measure was introduced to have a faster way to assess the effectiveness of the heuristic with the hypothesis that a decrease in permutation distance also implies a decrease in solve time variability. Although it seems a reasonable assumption, it is worth noting that it has been observed in the past that permutations of only a few columns can have an impact on solve time variability, so even in the case of a very close to zero permutation distance, some non-trivial variability in solve time can be expected. On the other hand, we can safely assume that whenever the heuristic makes the permutation distance greater instead of reordering towards a unique form, we can expect only to have equal or greater solve time variability.

When considering time, an extra layer of complexity comes up when having to analyze and compare experiment results. Solve time can be measured, and it is also given as output by most modern solvers, but it is always dependent on the hardware the solver is running on and CPU core availability. This makes it harder to have consistency of results across different runs and machines, and would require either always running the experiment on the same hardware or repeating the experiments on different machines to ensure that the results obtained are not dependent on the specific hardware. A feature

of the modern solvers that comes in handy to address this issue, is having the solve time expressed both in seconds and in work units [11]. In contrast to runtime, work is deterministic, meaning that you will get the same result every time provided you solve the same model on the same hardware with the same parameter and attribute settings. There will still be differences between machines when they have different processors, but there will be more consistency when measuring variability on the same machine, which will be our choice for the experiment.

2.4. Evaluation Methodology

The two metrics described above, permutation distance and solve time, particularly when measured in work units, will be used throughout the experimental part of this thesis to assess the effectiveness of the reordering heuristic. In particular, we aim to investigate whether lower permutation distances correlate with more stable solve times. By jointly analyzing these metrics, we will be able to quantify both the structural consistency introduced by the heuristic and its practical impact on solver performance.

3 | Exploring Reordering Approaches: From Decomposition to Heuristics

To build the heuristic that will be applied to problem instances with the aim of obtaining a unique reorder starting from any permutation of it, a great starting point may be analyzing similar procedures within solver algorithms and other problem decomposition strategies already explored in the literature.

3.1. Block decomposition

Building a block-diagonal form [2] is one approach that can be found in literature and has been successfully applied to matrices to create reorderings of them. This strategy is applicable whenever the considered matrix is sparse, and it is possible to arrange rows and columns to create a diagonal structure with blocks of non-zero coefficients on the diagonal, as shown in Figure 3.1.

Researchers have tried to leverage automatic decomposition techniques to come up with such block-diagonal structures and exploit them mainly to achieve speedups in the solver runtime, meaning these efforts were not optimized to reduce variability. One reason is that those decomposition experiments started 20 years ago, while variability came up in papers only starting from 2013 [15]. Finding a block-diagonal structure can also be interpreted as partitioning into independent blocks with only a small set of linking constraints, so that it is possible to treat each block as a separate problem that can be solved individually and coordinated via the few linking constraints.

We can find similar structures through an adjacency matrix transformation, which consists of constructing a graph representation of the problem's matrix, from which it is visible that some portions of the graph are more connected than others. It is then possible to apply some reordering to make the clusters of vertices denser and reduce the edges between

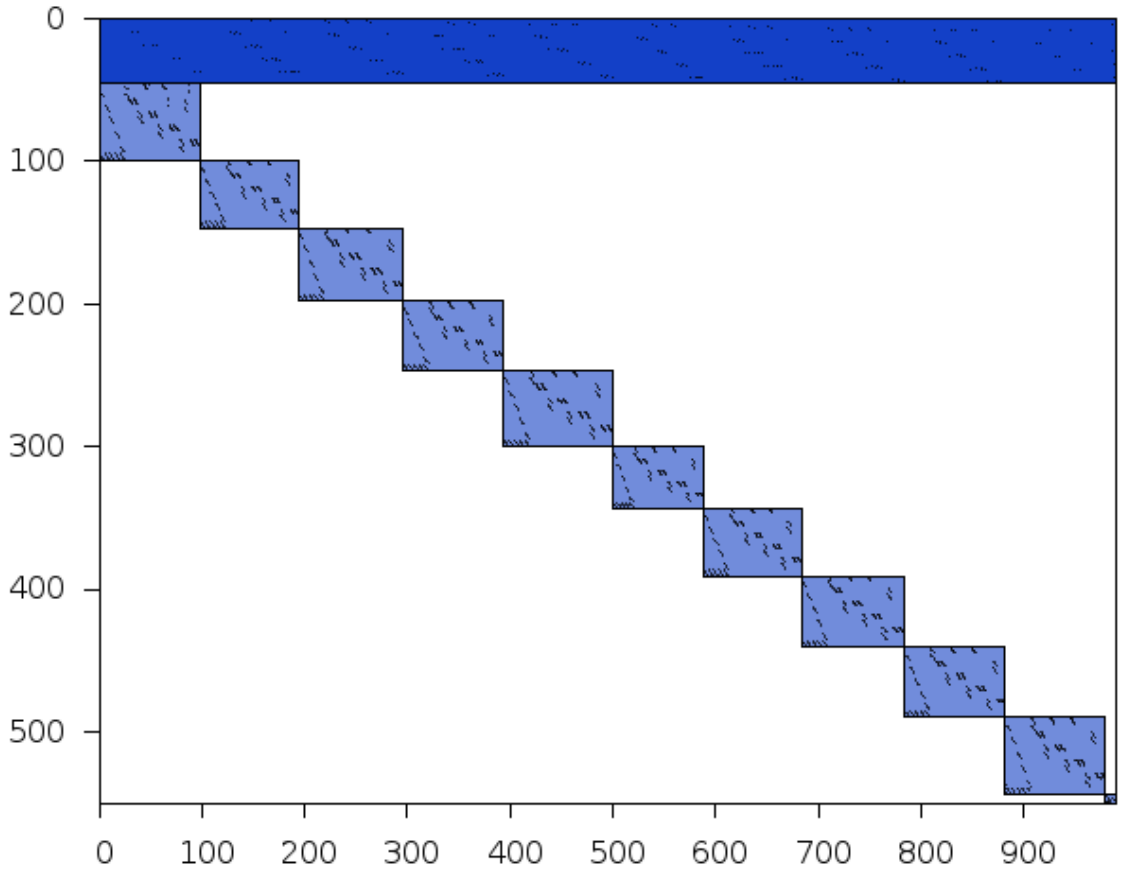

Figure 3.1: Decomposed Structure of the MIP Instance exp-1-500-5-5. This figure illustrates the block decomposition structure of the original MIP instance exp-1-500-5-5, as reported in the MIPLIB dataset. The image highlights the underlying sparsity pattern of the constraint matrix, where rows represent constraints and columns represent variables. Non-zero coefficients in the matrix are marked and represented as black points

these groups. This will reflect in a block-diagonal structure as well. Recently, hypergraph partitioning has emerged as an evolution of graph partitioning, the main difference is that it is more effective as it captures more precisely the relationships between variables and constraints. Similarly as the simple graph method, it rearranges the matrix with the aim to reduce coupling between clusters of nodes. Some famous implementations of these techniques include Benders decomposition [4] and Dantzig-Wolfe decomposition [3] within branch-and-price frameworks.

Even though they may seem great starting points to create a unique structure from problem instance variations, the granularity of the resulting block is not as fine as we need. In our case, we cannot have a too coarse granularity with big variables subblocks, but instead, we need a very fine structure that we can use to apply the ordering. In the

majority of practical applications of these techniques, it is not an issue because the aim is to simplify the structure into smaller independent and weakly coupled submatrices. For example, in optimization they can be successfully applied to network design problems as a way to identify clusters or communities of nodes within the system.

3.2. Rule-based heuristic

Now that we have a fundamental knowledge of how the problems are structured, what metrics we are considering to measure the effectiveness of our heuristic, and what is already existing in the field along with its characteristics, we can start to design our approach.

We define a score-based method, where a score is given to every variable and constraint of the problem matrix. Once the score has been computed for all problem parameters, they are sorted in ascending order, and that will be our heuristic algorithm applied to different permutations of the problem, which will hopefully lead to the exact same form only when we are lucky, but at least it will lead to a considerable decrease in variability. A simple way to build such a scoring function is to add a term to it for each piece of information we can extract from the problem instances. Each of these terms will be translated into a rule, and the composition of all the rules will be used as the final scoring function. Based on how we structure the rule, it may happen that the effectiveness of the score is not dependent only on the rule content, but also on the ordering between rules.

4 | Design and Application of Reordering Rules

4.1. Hierarchical Rules

In the beginning, we will not consider the impact of scaling at all, even though the ambition is to come up with rules that are scale-invariant. The purpose of these initial rules is to lay a foundation to build upon to achieve our goals and also to roughly estimate the boundaries of our improvements. As we need to reorder both variables and constraints, we will build two scoring functions with similar structures: one to reorder columns and one to reorder rows.

The rules composing those scoring functions are responsible for creating a multi-level block decompostion, where variables and constraints are first categorized into blocks based on some common features. Within each block, secondary criteria will be applied to further separate variables. Since at the end we are trying to aggregate the block ordering to build a score, one option we have would be that each of these levels of blocks is represented in the score function as a term multiplied by a power of ten, ensuring each blocking rule has an impact on a different order of magnitude of the score. Even though it may be an effective solution when having a few terms, sometimes summing up the terms leads to unexpected ties in scores. To have a more robust ordering, each rule is applied and its score is inserted into a tuple, which is later lexicographically sorted respecting the priority between blocking rules.

Column (Variable) Ordering

The ordering of variables is determined using a hierarchical block structure.

Block Score: Each variable i is first assigned a block score based on structural characteristics:

$$\text{BlockScore}_i = (P_{\text{type}}(i), P_{\text{bounds}}(i)) ,$$

where:

- $P_{\text{type}}(i)$ assigns a priority label based on variable type: binary, integer, continuous.
- $P_{\text{bounds}}(i)$ assigns a priority category to each variable based on the characteristics of its bounds:
  - Variables with both bounds finite and either entirely nonnegative (lower bound ≥ 0) or entirely nonpositive (upper bound ≤ 0).
  - Variables with both bounds finite but straddling zero (i.e., lower bound $< 0 <$ upper bound).
  - Variables with exactly one bound infinite (either lower or upper bound is infinite).
  - Variables with both bounds infinite (unbounded in both directions).

Intra-Block Score: Within each block, a secondary intra-block score is computed as:

$$\text{IntraBlockScore}_i = \left(\sum_{j=1}^{m} \log(1 + |a_{ji}|), \log(1 + |c_i|), \#\text{occurrences}_i \right),$$

where:

- $\sum_{j=1}^{m} \log(1 + |a_{ji}|)$ is the sum of logarithmic-scaled absolute coefficients across all constraints.
- $\log(1 + |c_i|)$ encodes the influence of the variable in the objective function.
- $\#\text{occurrences}_i$ counts the number of constraints in which the variable appears.

Final Ordering: The final score tuple used for ordering is:

$$\text{Score}_i = (\text{BlockScore}_i, \text{IntraBlockScore}_i),$$

where the sorting is performed lexicographically:

$$\text{Sort}_{\text{lex}} (\text{Score}_i).$$

Row (Constraint) Ordering

The ordering of constraints follows a similar hierarchical approach.

Block Score: Each constraint j is first assigned a block score:

$$\text{BlockScore}_j = (P_{\text{sense}}(j), P_{\text{composition}}(j)),$$

where:

- $P_{\text{sense}}(j)$ assigns a priority label based on the constraint sense: greater than or equal ($\geq$), equality ($=$), less than or equal ($\leq$).
- $P_{\text{composition}}(j)$ assigns a priority label based on coefficient types: only integral, only continuous, mixed.

Intra-Block Score: Within each block, an intra-block score is computed as:

$$\text{IntraBlockScore}_j = \left(\sum_{i=1}^{n} \log(1 + |a_{ji}|), \log(1 + |b_j|), \log(1 + \text{Range}_j) \right),$$

where:

- $\sum_{i=1}^{n} \log(1 + |a_{ji}|)$ is the sum of logarithmic-scaled absolute coefficients across all variables.
- $\log(1 + |b_j|)$ accounts for the numerical importance of the right-hand side value.
- $\log\left(1 + \text{Range}_j\right)$ encodes the variability of the coefficients in the constraint, where the *range* Range_j is defined as the difference between the maximum and minimum coefficient values in row j of matrix A, i.e.,

  $$\text{Range}_j = \max_{1 \leq i \leq n} a_{ji} - \min_{1 \leq i \leq n} a_{ji}.$$

  where a_{ji} denotes the coefficient of variable i in constraint j, which may be positive, negative, or zero.

Final Ordering: The final score tuple used for ordering is:

$$\text{Score}_j = (\text{BlockScore}_j, \text{IntraBlockScore}_j),$$

where constraints are sorted lexicographically:

$$\text{Sort}_{\text{lex}}\left(\text{Score}_j\right).$$

Tables of Typical Ranges

Tables 4.1 and 4.2 report the typical ranges of scoring terms for columns (variables) and rows (constraints), with some values shown after applying logarithmic scaling.

Because logarithms are applied prior to scoring, the values remain in comparable numerical ranges and avoid numerical explosions that could skew the ordering process.

Column (Variable) Scoring

Term	Typical Min	Typical Max
Type Priority (P)	1	3
Bounds Priority	1	3
Sum of Coefficients	0	276,310
Objective Coefficient	0	2,763
Occurrences in Constraints	0	100

Table 4.1: Typical minimum and maximum values for each scoring term used in evaluating problem columns (variables). These ranges reflect observed data after applying logarithmic scaling to maintain numerical stability in the scoring process.

Row (Constraint) Scoring

Term	Typical Min	Typical Max
Sense Priority (P)	1	3
Composition Priority	1	3
Sum of Coefficients	0	276,310
RHS Value	0	2,763
Range Value	0	2,763

Table 4.2: Typical minimum and maximum values for scoring terms used in evaluating problem rows (constraints). Logarithmic scaling has been applied to the raw values to keep them within comparable numerical ranges and avoid distortion during sorting.

Overall, these typical ranges clarify how the block score and intra-block score differ in scale and nature. The ordering relies on comparing these two scores, with the block score providing a coarse prioritization and the intra-block score refining the order within each block.

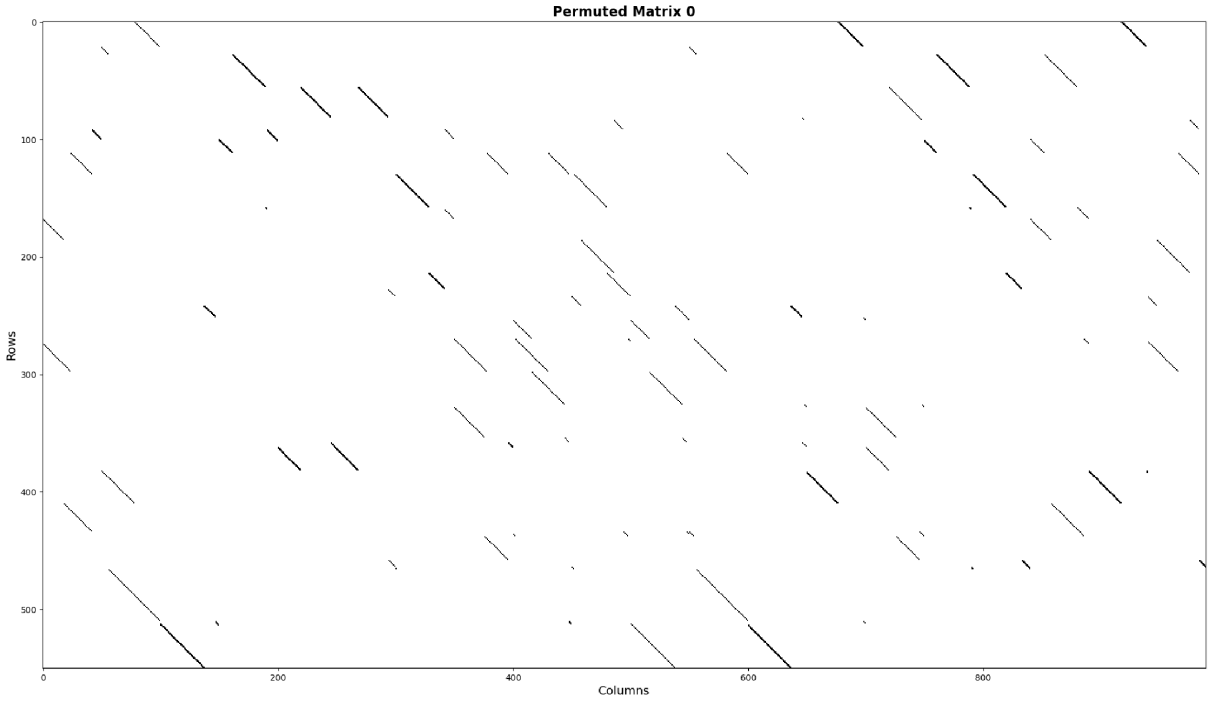

Figure 4.1: Initial view of the first permuted matrix instance of `exp-1-500-5-5`. The block structure is obscured due to the random permutation of rows and columns

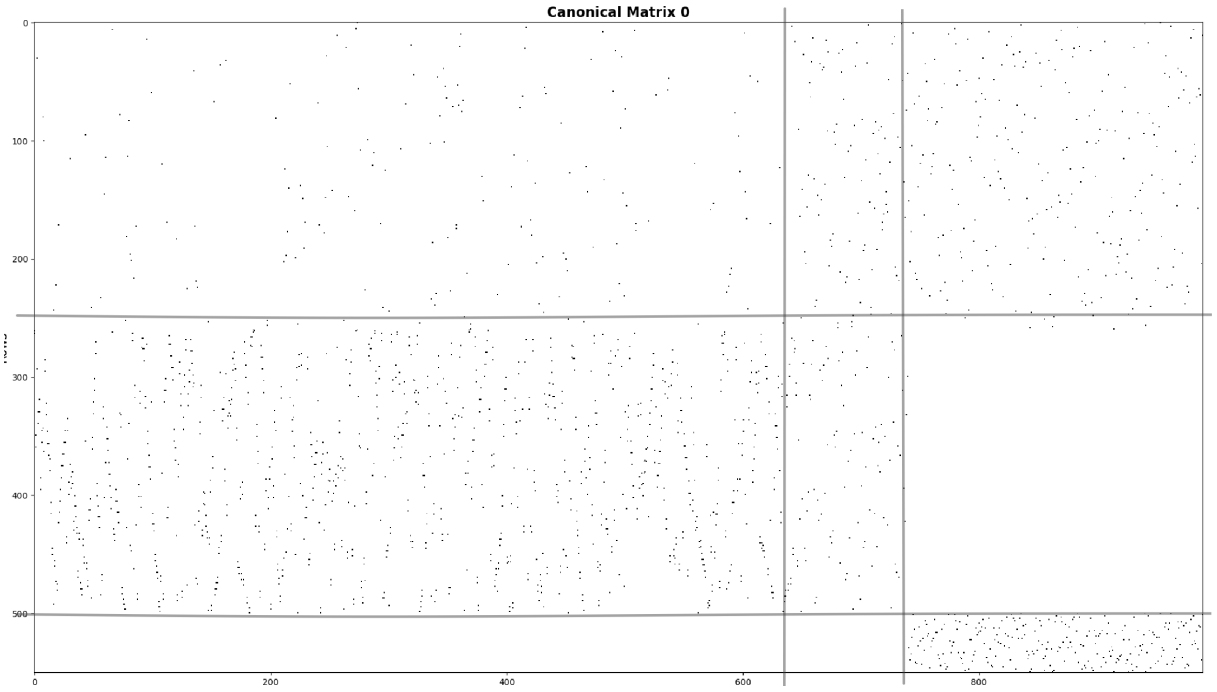

Figure 4.2: Reordered version of the first permutation shown in Figure 4.1. After applying the hierarchical reordering rules, we observe a clear emergence of structural blocks, reflecting the latent organization of the original matrix

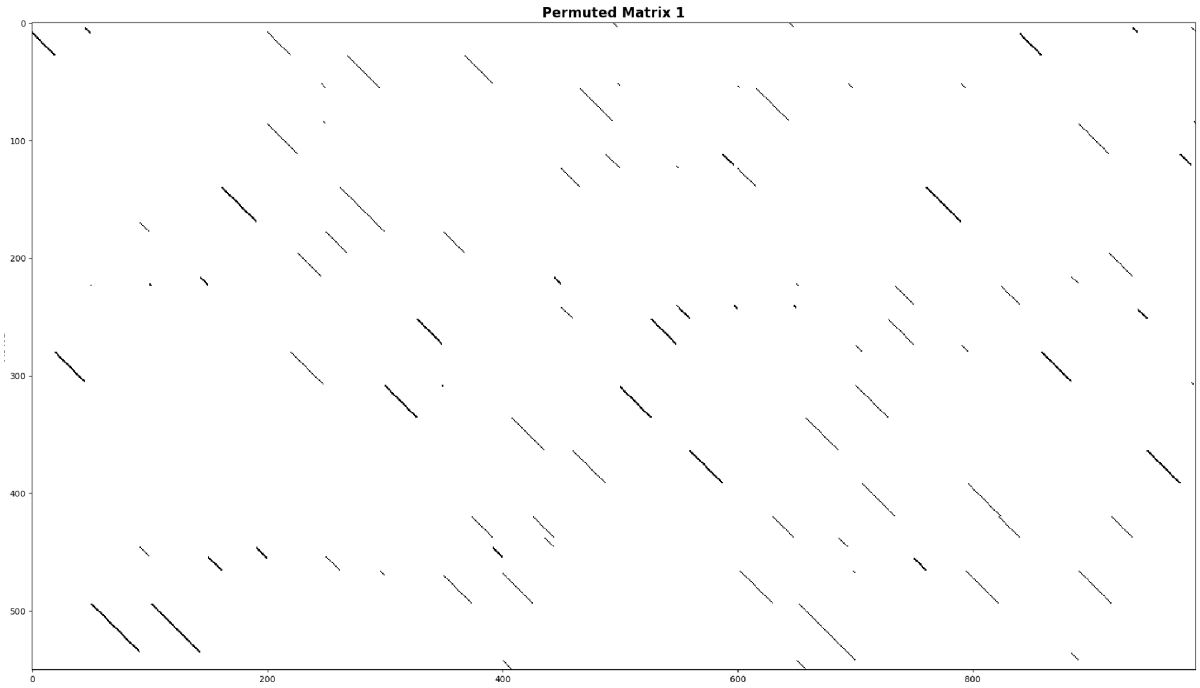

Figure 4.3: Initial view of a second permutation of the same matrix instance. The natural block structure is masked by the applied random permutation

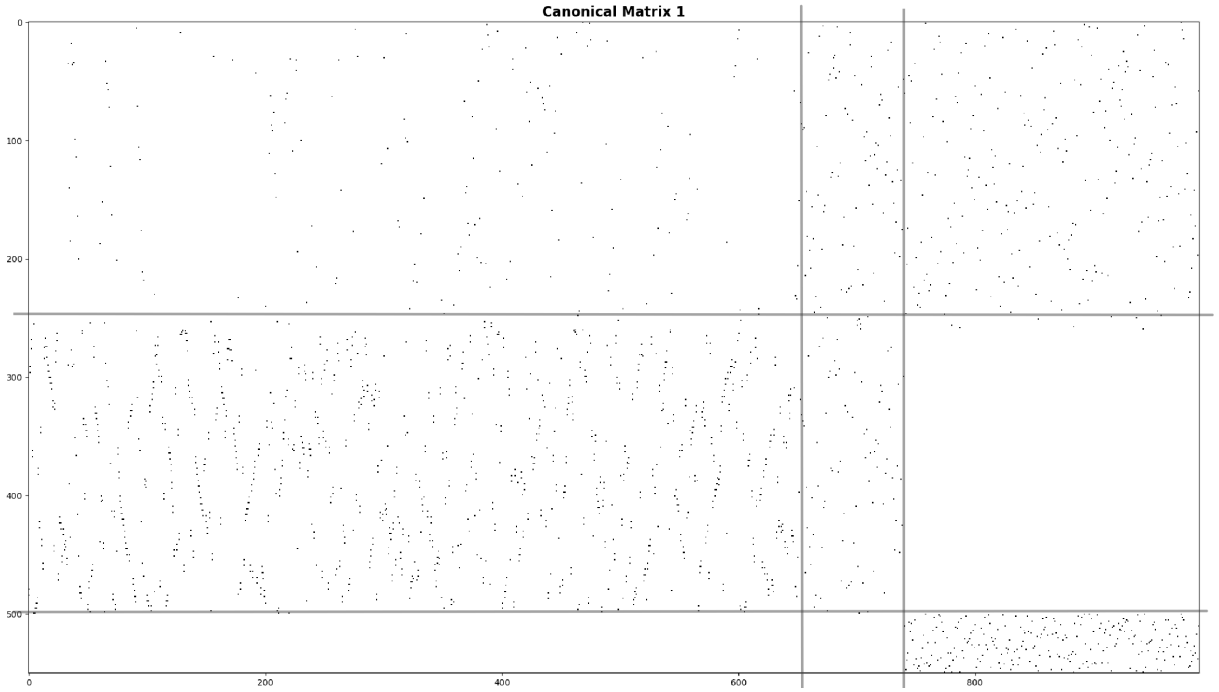

Figure 4.4: Reordered version of the second permutation shown in Figure 4.3. Despite the difference in the initial permutation, the hierarchical rules successfully recover a block structure similar to that of the first reordered matrix, confirming the robustness and consistency of the method

Rules Application Example

Figures 4.1 to 4.4 illustrate the impact of our hierarchical reordering rules on two distinct permutations of the same matrix instance. Each permutation is visualized before and after applying the rules. This highlights how structural regularity is recovered through reordering, even when the input matrices differ due to permutation.

4.2. Scale Invariance

Even though the previous rules seem to be effective, they are not as robust as we wish. The final goal of the reordering method is to obtain a close to unique ordering for each variation of the problem instance. The previous rules are effective when problems are only permuted, but sometimes scaling can also happen. It is sufficient for example that a constraint is multiplied by negative one or by two, and some of the rules, which are based on the coefficients' absolute value or the constraint sense, can end up producing a different ordering.

When talking about scale invariance, there are two possible ways a problem can be scaled. One is constraint scaling which means multiplying all row coefficients by a value and then adjusting the right-hand-side value accordingly so that the equation expresses the exact same constraint. The other one is similar, but instead of multiplying rows it consists in multiplying columns and adjusting the objective value coefficient.

This explains why cases where rules creating an ordering based on the absolute value of coefficients may lead to different results. After a proper sequence of scaling on rows and columns, any matrix coefficient can be virtually transformed to any value.

Note that when we apply a negative scaling, the constraint sense needs to be changed, which implies that rules creating a hierarchy on constraint sense are not robust when the problem is scaled since all "$\geq$" constraints can be easily transformed to "$\leq$" applying a negative scaling term to the corresponding rows.

Normalization strategies

To address the issue of scale invariance, where arbitrary scaling of coefficients can interfere with reordering rules, we explore normalization strategies. The idea is that by transforming all problem instances to a comparable scale, we can make structural patterns invariant to scaling and thus detectable by our reordering rules. We can be sure that after the normalization steps are applied, all the instances will have the same coefficients. In principle, applying scaling to both rows and columns, keeping into account the differences in variable types, and adjusting the objective function and the right-hand side

coefficients, seems doable and may intuitively be a good way to come up with a unique normal from, starting from different scaling of the same problem instance. This kind of transformation will also help the solvers, providing them with problems with greater numerical stability.

4.2.1. ℓ_2 Normalization for Continuous Variables

Let $A \in \mathbb{R}^{m \times n}$ be the coefficient matrix of a MILO problem. Define x_i as a continuous variable:

$$x_i \in \mathbb{R}, \quad \forall i \in \mathcal{C},$$

where $\mathcal{C}$ is the set of continuous variables.

Step 1: Compute the ℓ_2 Norm for Each Column

For each continuous variable x_i, let $a_{\cdot,i}$ denote the vector of nonzero coefficients in the i-th column of the constraint matrix:

$$a_{\cdot,i} = \{a_{j,i} \mid j \in \mathcal{J}_i,\, a_{j,i} \neq 0\},$$

where $\mathcal{J}_i = \{j : a_{j,i} \neq 0\}$ is the set of indices of constraints with nonzero coefficients for variable x_i. Then, compute the ℓ_2 norm of this vector as

$$\|a_{\cdot,i}\|_2 = \sqrt{\sum_{j \in \mathcal{J}_i} a_{j,i}^2}.$$

Step 2: Define the Scaling Factor

$$s_i = \frac{1}{\|a_{\cdot,i}\|_2}.$$

Step 3: Scale the Coefficients

$$c_i \leftarrow \frac{c_i}{s_i}, \quad a_{j,i} \leftarrow a_{j,i} \cdot s_i, \quad \forall j \in \mathcal{J}_i.$$

4.2.2. GCD Normalization for Discrete Variables

Let x_i be a discrete variable:

$$x_i \in \mathbb{Z} \quad \text{or} \quad x_i \in \{0, 1\}, \quad \forall i \in \mathcal{D},$$

where $\mathcal{D}$ is the set of discrete variables (integer or binary).

Step 1: Select Integer Coefficients Only Let $\mathcal{A}_i^{\mathbb{Z}} \subseteq \{a_{j,i} : j \in \mathcal{J}_i\}$ be the subset of coefficients that are integer-valued:

$$\mathcal{A}_i^{\mathbb{Z}} = \{a_{j,i} \in \mathbb{Z} \mid j \in \mathcal{J}_i\}.$$

Step 2: Compute the Greatest Common Divisor (GCD) Compute the greatest common divisor g_i of the absolute values of the integer coefficients:

$$g_i = \gcd\left(|a_{j,i}| : a_{j,i} \in \mathcal{A}_i^{\mathbb{Z}}\right).$$

This step ensures that only coefficients that are exactly integer-valued contribute to the normalization factor, which is important to avoid corrupting the structure of the model due to rounding or the inclusion of fractional values.

Step 3: Define the Scaling Factor

$$s_i = \frac{1}{g_i}.$$

Step 4: Scale the Coefficients

$$c_i \leftarrow \left\lfloor \frac{c_i}{s_i} + \frac{1}{2} \right\rfloor, \quad a_{j,i} \leftarrow a_{j,i} \cdot s_i, \quad \forall j \in \mathcal{J}_i.$$

Step 5: Ensure Integrality Ensure that the transformed coefficients maintain integrality:

$$c_i \in \mathbb{Z}, \quad a_{j,i} \in \mathbb{Q}.$$

4.2.3. Row Normalization

Each row j in the constraint matrix is normalized differently based on the presence of discrete variables.

Case 1: Row Contains Discrete Variables If row j contains at least one discrete variable $x_i \in \mathcal{D}$, define the row scaling factor using the GCD of the integer-valued discrete coefficients:

$$r_j = \frac{1}{\gcd\left(|a_{j,i}| : i \in \mathcal{D} \cap \mathcal{I}_j,\ a_{j,i} \in \mathbb{Z}\right)},$$

where $\mathcal{I}_j$ is the set of variables appearing in row j. Only coefficients that are both associated with discrete variables and integer-valued are considered.

Case 2: Row Contains Only Continuous Variables If row j contains only continuous variables, compute the ℓ_2 norm of the row:

$$r_j = \frac{1}{\|a_{j,\cdot}\|_2} = \frac{1}{\sqrt{\sum_{i \in \mathcal{C} \cap \mathcal{I}_j} a_{j,i}^2}}.$$

Apply Row Scaling All coefficients in row j, including the right-hand side b_j, are updated as:

$$a_{j,i} \leftarrow a_{j,i} \cdot r_j, \quad b_j \leftarrow b_j \cdot r_j, \quad \forall i \in \mathcal{I}_j.$$

4.2.4. Building the Normalized Model

Once all variables and rows are scaled, a new MILO model is reconstructed using the transformed data:

1. **Variable Creation**: Recreate variables with normalized bounds and preserved types.
2. **Objective Function Update**: Insert the normalized objective function using updated c_i.
3. **Constraint Reconstruction**: Add constraints using the normalized coefficient matrix A' and updated right-hand sides b'.

Even if it seems to be a great approach at first sight, this method has several downsides. First, when applying divisions by the ℓ_2 norm we actually contribute to numerical instability caused by floating point operations and chains of rounding and approximating. However, a more fundamental drawback is that normalization does not guarantee a unique final form when starting from different initial scales. Often, if we consider one single coefficient in the problem matrix, we can think of several scalings applied in different orders to rows and columns that leads to the same number. As a result, two equivalent formulations of the same problem that were just scaled differently may be normalized by our procedure, but the result will still have some differences in coefficients. At this point, all our normalization efforts are useless in the first place as the reordering rules cannot rely on the hypothesis that scale invariance is not important as it cannot be effectively removed by our normalization.

Example: Equivalent Models Leading to Different Normalized Forms

We present two mathematically equivalent MILO models, Model A and Model B, that differ only in their coefficient scaling. Both are normalized using column and row normalization as defined in the previous section, yet they yield different normalized forms. This shows that normalization does not ensure a canonical representation.

Model A (Original)

$$\begin{aligned} \min \quad & 2\,x_1 \quad + 6\,x_2 \\ \text{s.t.} \quad & 4\,x_1 \quad + 2\,x_2 \quad \leq 8 \quad \text{(Row 1)}, \\ & \qquad\quad 0.2\,x_2 \quad \leq 1 \quad \text{(Row 2)}, \\ & \mathrm{x}_1 \in \mathbb{Z}, \quad x_2 \in \mathbb{R}. \end{aligned}$$

Model B (Scaled)

$$\begin{aligned} \min \quad & 3\,x_1 \quad + 9\,x_2 \\ \text{s.t.} \quad & 6\,x_1 \quad + 3\,x_2 \quad \leq 12 \quad \text{(Row 1)}, \\ & \qquad\quad 0.3\,x_2 \quad \leq 1.5 \quad \text{(Row 2)}, \\ & \mathrm{x}_1 \in \mathbb{Z}, \quad x_2 \in \mathbb{R}. \end{aligned}$$

Normalization of Model A

Column-normalization:

$$\begin{aligned} \text{Discrete } x_1: \quad & \gcd(|4|) = 4 \;\Rightarrow\; s_1 = \tfrac{1}{4}, \quad c_1 = \tfrac{2}{s_1} = 8, \quad a_{1,1} = 1. \\ \text{Continuous } x_2: \quad & \|a_{\cdot,2}\|_2 = \sqrt{2^2 + 0.2^2} = \sqrt{4 + 0.04} = \sqrt{4.04} \approx 2.00997, \\ & s_2 \approx 0.4975, \quad c_2 = \tfrac{6}{s_2} \approx 12.06, \\ & a_{1,2} = 2 \cdot s_2 \approx 0.995, \quad a_{2,2} = 0.2 \cdot s_2 \approx 0.0995. \end{aligned}$$

Row-normalization:

$$\begin{aligned} \text{Row 1:} \quad & \gcd(1,\ 0.995) \text{ (treated as numeric)} \Rightarrow r_1 = 1, \quad b_1 = 8. \\ \text{Row 2:} \quad & \|a_{2,\cdot}\|_2 = 0.0995 \Rightarrow r_2 = \tfrac{1}{0.0995} \approx 10.05, \\ & a_{2,2} = 1, \quad b_2 = 1 \cdot 10.05 = 10.05. \end{aligned}$$

Final normalized Model A:

$$\begin{aligned} \min \quad & 8\,x_1 \quad + 12.06\,x_2 \\ \text{s.t.} \quad & x_1 \quad + 0.995\,x_2 \quad \leq 8, \\ & \qquad\; x_2 \qquad\;\; \leq 10.05, \\ & \mathrm{x}_1 \in \mathbb{Z}, \quad x_2 \in \mathbb{R}. \end{aligned}$$

Normalization of Model B

Column-normalization:

$$\begin{aligned} \text{Discrete } x_1: \quad & \gcd(|6|) = 6 \Rightarrow s_1 = \tfrac{1}{6}, \quad c_1 = \tfrac{3}{s_1} = 18, \quad a_{1,1} = 1. \\ \text{Continuous } x_2: \quad & \|a_{\cdot,2}\|_2 = \sqrt{3^2 + 0.3^2} = \sqrt{9 + 0.09} = \sqrt{9.09} \approx 3.0166, \\ & s_2 \approx 0.3317, \quad c_2 = \tfrac{9}{s_2} \approx 27.13, \\ & a_{1,2} = 3 \cdot s_2 \approx 0.995, \quad a_{2,2} = 0.3 \cdot s_2 \approx 0.0995. \end{aligned}$$

Row-normalization:

$$\begin{aligned} \text{Row 1:} \quad & \gcd(1,\ 0.995) \Rightarrow r_1 = 1, \quad b_1 = 12. \\ \text{Row 2:} \quad & \|a_{2,\cdot}\|_2 = 0.0995 \Rightarrow r_2 = \tfrac{1}{0.0995} \approx 10.05, \\ & a_{2,2} = 1, \quad b_2 = 1.5 \cdot 10.05 = 15.08. \end{aligned}$$

Final normalized Model B:

$$\begin{aligned} \min \quad & 18\,x_1 \quad + 27.13\,x_2 \\ \text{s.t.} \quad & x_1 \quad + 0.995\,x_2 \quad \leq 12, \\ & \qquad\; x_2 \qquad\;\; \leq 15.08, \\ & \mathrm{x}_1 \in \mathbb{Z}, \quad x_2 \in \mathbb{R}. \end{aligned}$$

Although Model A and Model B describe the same feasible region and objective direction, their normalized forms differ due to scaling in the original formulation. This highlights that normalization alone is not sufficient to identify equivalent instances when coefficients differ by scale. Therefore, reordering rules must be scale-invariant and not rely on the magnitude of coefficients to reliably obtain a reordered form.

4.3. Recursive Rules

To be properly scale-invariant, rules have to recognize structural patterns that are not dependent on scale. For example, rules that count nonzero entries in the matrix are much more robust, since scaling by 0 is not a valid transformation, as it does not lead to an equivalent form. Because of that we can safely assume it will never happen.
On the other hand, such rules are only a few and ties are far more likely to happen if we decide to simply apply them as before. That is because they are not based on coefficient values, which are far more likely to produce a different score for each variable and constraint.
Something we can observe about the previous hierarchical ordering is that the rules can be applied to the whole matrix and, for each rule, the score of a specific variable does not change based on the order in which rules are applied. This is because these rules are only considering properties of the variables themselves, and not other information about the matrix they are part of. If we decide to rely on structural properties, which are influenced by multiple variables within the matrix, we are going to have different results applying rules in a different order.
In this case, rules will be applied recursively to the submatrices that are created at every iteration, until no further decomposition is found. Of course, at the beginning there will be only one matrix, which is the original.

Primary and Secondary Block Rules

The process begins with primary (parent) block rules, which attempt to partition the full index set based on fundamental characteristics like variable type (binary, integer, continuous) or constraint sense ($\leq, =, \geq$). These rules define coarse-grained partitions, which, when effective, ensure a strong structural hierarchy in the ordering process. If no primary rule is effective at a given recursion level, secondary (child) block rules are applied as a fallback, using alternative criteria such as coefficient magnitudes, bounds, or sparsity measures.

Formally, for an index set $\mathcal{V}$ (variables) or $\mathcal{C}$ (constraints), a rule r partitions the set as follows:

$$\mathcal{V} = \bigcup_{l \in \mathcal{L}_r} \mathcal{V}_l, \quad \text{where} \quad \mathcal{V}_l = \{v \in \mathcal{V} \mid r(v) = l\}.$$

A rule is considered effective if $|\mathcal{L}_r| > 1$, meaning it successfully partitions the set.

Recursive Decomposition

Once a partition is established, the recursive process continues within each sub-block. The recursion follows these steps:

1. **Apply Primary Rules:** Attempt to partition the block using primary rules. If an effective partition is found:
$$V_k = \bigcup_{l \in \mathcal{L}_r} V_{k,l}.$$
The procedure recurses within each subset $V_{k,l}$.
2. **Fallback to Secondary Rules:** If no primary rule is effective, secondary rules are applied. The order of child rules is rotated across recursive calls to prevent bias.
3. **Termination:** Recursion stops when no further partitioning is possible, because a block cannot be subdivided any further according to the rules, or because the maximum recursion depth has been reached.

Intra-Block Ordering

When recursion terminates, the indices within that block are ordered using intra-block rules. Each index v is assigned a **score tuple**:

$$\mathbf{s}(v) = (s_1(v), s_2(v), \ldots, s_k(v)) .$$

The final order is determined by lexicographical sorting:

$$\mathbf{s}(v_{(1)}) \leq_{\text{lex}} \mathbf{s}(v_{(2)}) \leq_{\text{lex}} \cdots \leq_{\text{lex}} \mathbf{s}(v_{(n)}).$$

Final Ordering

The final ordering is constructed by recursively applying:

$$\text{Sort}_{\text{Recursive}}(V) = \begin{cases} \bigcup_{l \in \mathcal{L}} \text{Sort}_{\text{Recursive}}(V_l), & \text{if } V \text{ is partitioned into blocks } \{V_l\}_{l \in \mathcal{L}}, \\ \text{LexSort}\big(V, \mathbf{s}(v)\big), & \text{otherwise.} \end{cases}$$

Recursive Rules Application Example

In this section, we demonstrate the impact of applying our recursive reordering rules to two distinct permutations of the same matrix instance. As before, each permutation is

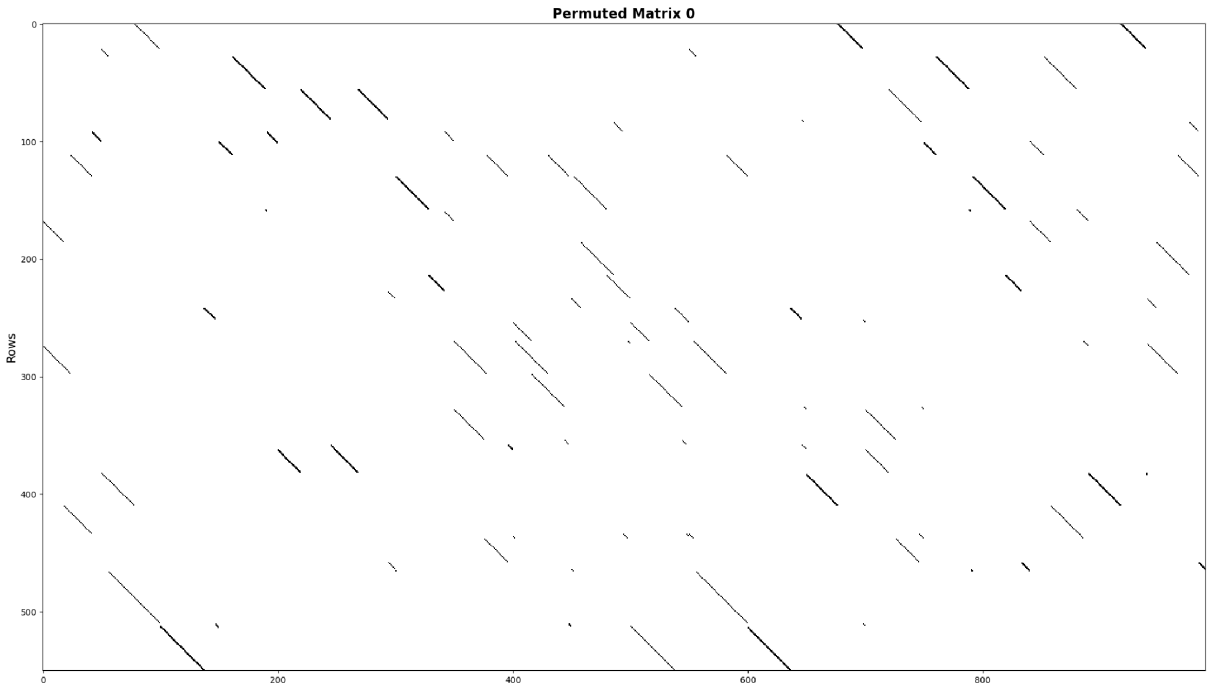

Figure 4.5: Initial view of the first permuted matrix instance of `exp-1-500-5-5`. The block structure is obscured due to a random permutation of rows and columns

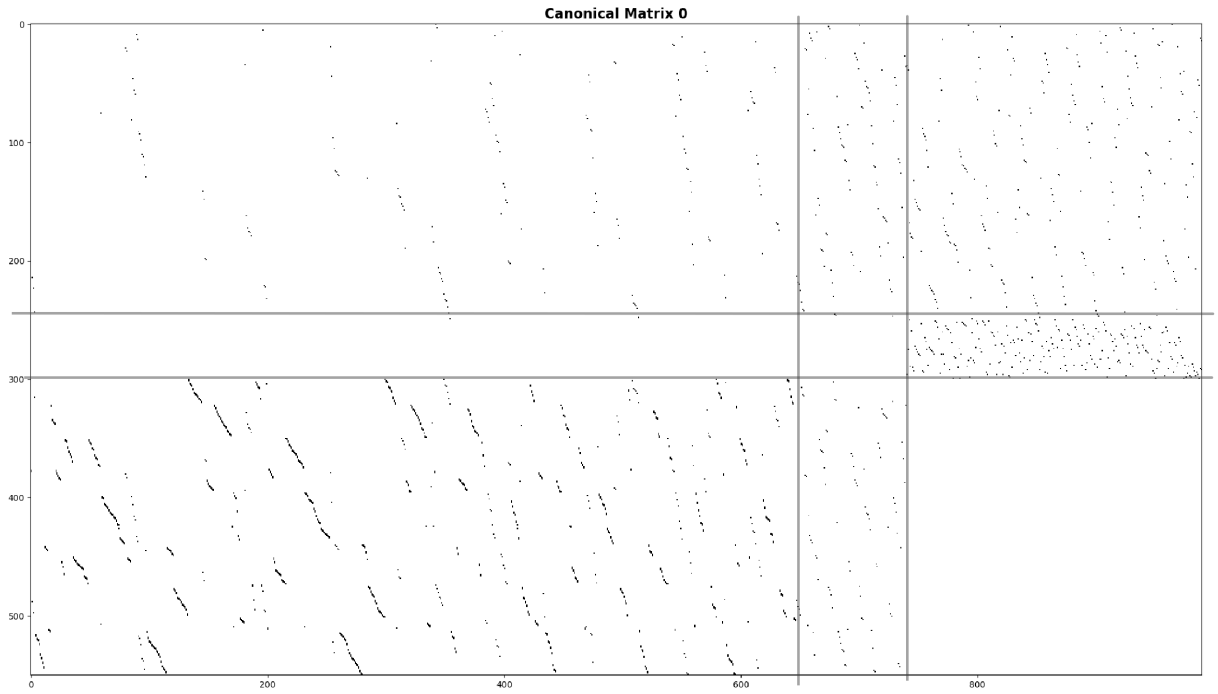

Figure 4.6: Reordered version of the first permutation using recursive rules. The recovered structure clearly shows a modular block arrangement, indicating successful restoration of the matrix's underlying pattern

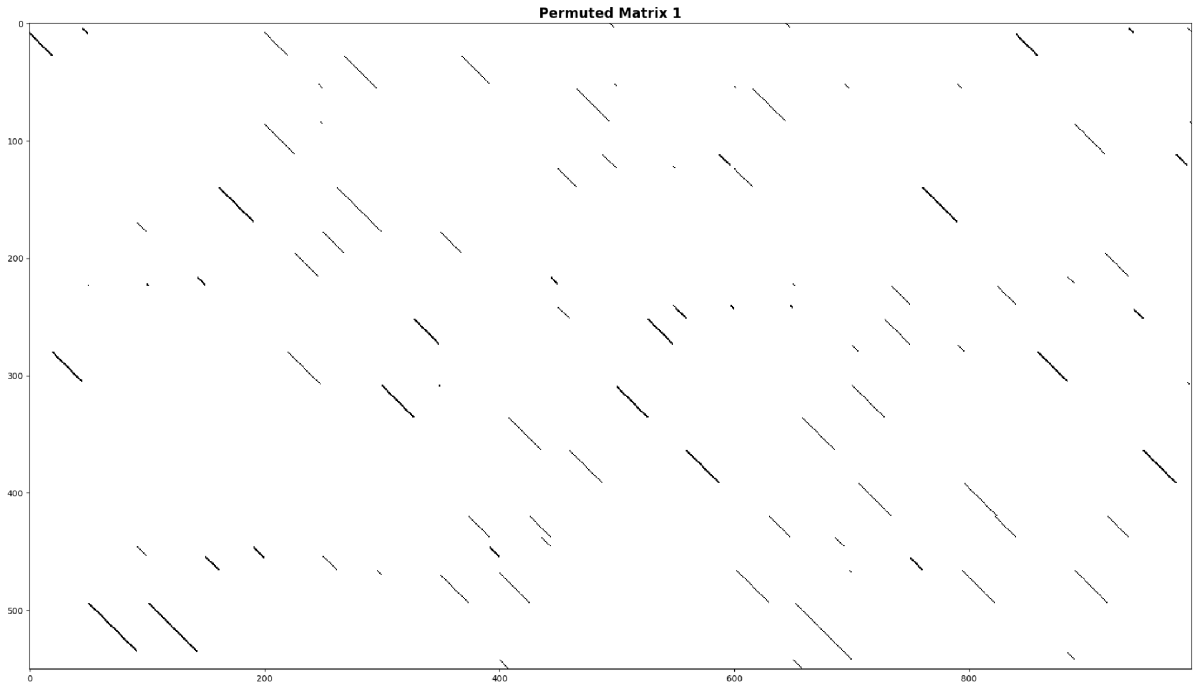

Figure 4.7: Initial view of a second permutation of the same matrix instance. The randomized order conceals the original matrix structure

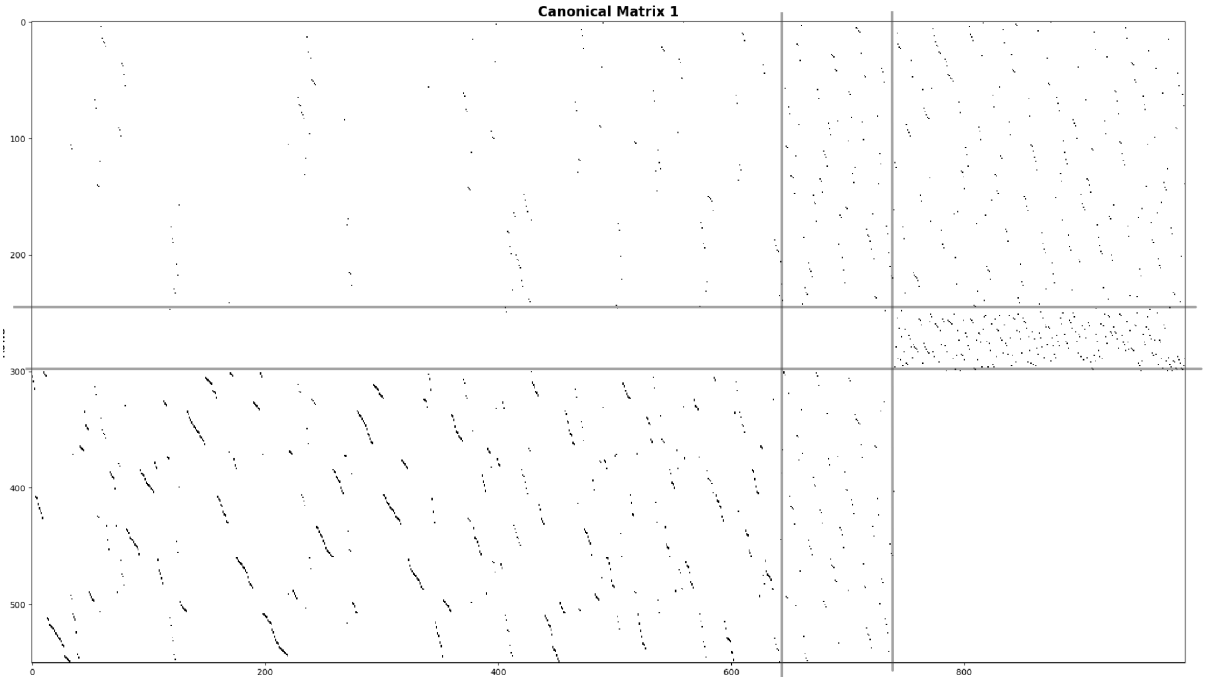

Figure 4.8: Reordered version of the second permutation using recursive rules. As with the first permutation, the recursive algorithm successfully restores a similar block structure, showing the robustness and generality of the method

shown both before and after rule application. Figures 4.5 to 4.8 highlight the consistency and effectiveness of recursive rules in revealing latent block structures, regardless of the specific input permutation.

Granularity

One important feature to consider when evaluating the effectiveness of a recursive block decomposition algorithm, as we have just described, is granularity. In fact, as we mentioned before, the existing block decomposition methods are not suitable to our purpose because they create structures of submatrices with a too coarse granularity. With our method, we aim to find a set of rules that once applied recursively achieve a very fine granularity so that ties are minimized, and we can have a very precise ordering within the matrix. Intuitively, we can expect to have a high correlation between how effective the rules are reducing our permutation distance and solving time variability and how granular is the decomposition we obtain after applying the rules, using the recursive method.

4.4. Rules ordering

While the ordering of rules is not important in the case of the hierarchical approach, since each rule's contribution is simply summed to produce a score which does not depend on the subblocks sizes, in the recursive approach, the way subblocks are generated at the beginning of the algorithm can significantly impact the final score, as some rules depend on counts within the submatrix. To be as consistent as possible across the possible problem permutations, first we will apply structure detecting rules that are not based on submatrix item numbers to have an initial subdivision. One example is divisions according to variable type, which is a structural feature, as no scaling can change it. After that, all the rules based on counts should be tested in different orders, trying to understand which order leads to the best granularity.

We may need to define a set of rules to deal with particularly difficult problems which are not recognized by our algorithm. For example, in set covering and set packing problems variables and constraints are not heterogeneous enough, as they are all binary. We need to define some very specific rules to apply to these problems that should not have the unwanted effect of isolating parts of the matrix that will be better detected by other rules. To deal with this issue we will place these specific rules at the end of the application chain. This way whenever we find a problem that was not subdivided effectively by the other rules, such as an instance of a set packing problem, we will be able to detect its structures. On the other hand, more generic problems at that point of the recursive application will

be already subdivided with fine granularity, and the final specific rules will have an impact only on submatrices that have actually the right characteristics.

5 | Experimental Setup

After designing the algorithm and defining the metrics that will be used to evaluate if the algorithm is successful in reducing variability, we need to define the methodology we will use to conduct experiments on a set of known problems, that should be large enough to come with some conclusion and can be replicated to confirm them or try to do further refinements.

5.1. MIPLIB

The sixth Mixed Integer Programming Library (MIPLIB 2017) [19] is a set of known problems that are widely used in research. It contains 1065 optimization instances, which are different along several key dimensions such as size, solving difficulty, application domain and structural features. There is a subset of 240 "benchmark" instances [18] that is representative of the whole library and that we can use to study solver performance. Each instance of the benchmark set is reasonably difficult, has a consistent optimal solution and has stable numerical properties. This way outliers are not included in the set and we have a fair representation of most real-world cases.

Most of the problems in the benchmark are tagged as easy, meaning they are solvable within one hour on common hardware. Instances are of a heterogeneous size and have also other tags indicating particular structural properties such as "knapsack" or "mixed-integer". Each problem is also named according to the domain it was extrapolated from, so we have a range of instances coming from common fields such as production planning and network design. All those characteristics make this set ideal to collect data about experiments and draw some conclusions based on some statistics.

5.2. Testing Environment

To conduct our experiments we need an environment where we can import problems from the MIPLIB, apply to them some transformations and then test our reordering algorithms. At the end we will compare the results found with one of the common commercial solvers,

which in our case will be Gurobi [10], given its good integration with Python through the gurobipy [9] library and also the ease with which it is possible to extract the matrix representation of the problem, to apply our transformations and calculate the metrics defined earlier.
All the other environment features are built with Python, with the help of some linear algebra libraries called numpy [20] and scipy [22]. These libraries make it easy to manipulate matrices and vectors in a memory-efficient way.
One feature of MIP problems that has not been discussed in detail so far is their sparsity, meaning that most of the problems, when written in matrix form, are represented by matrices with few non-zero coefficients. This makes it easier to store them in a simplified representation, such as row-major format, that occupies far less memory than a simple Python vector or matrix and speeds up all computations.

5.2.1. Experimental Workflow

First, we will define the algorithm and rules we want to test. Then for all the instances we want to test, ideally all 240 of the MIPLIB, we will apply the following steps:

1. load the problem instance we want to test from one of the .mps files in the benchmark set of the MIPLIB 2017;
2. create an arbitrary number of permuted instances starting from it;
3. solve all the problems and calculate the metrics we are interested in;
4. apply the reordering algorithm to all the problems and calculate the same metrics;
5. compare the metrics before and after applying the problem.

Note that since in past research it was found that, to fully manifest the variability of a problem instance, it is enough to consider four variations of it, we can simply consider the original and three permutations of it to be able, after applying all the steps, to compare the results and say if an algorithm is effective or not.

Since the problems represent different real-world scenarios coming from different domains, we are not trying to find an approach that generalizes well when applied to all the 240 problems in the set, because there are some domains that are inherently more difficult for our algorithm. Our experiments will focus on finding an algorithm that performs well on a sufficiently big subset of the 240 problems, this way we will be able to build a unique form of them, starting from any variations, and that may eventually be used to optimize the solver performances for this particular subset. Our environment will need another

part that will be used to post-process all the data gathered through experiments and calculate the statistics that will be used to draw conclusions. Also in this part, we will continue using Python and some popular libraries to build visual representations of data like pandas [24], matplotlib [17] and seaborn [25]. Since the instances of the MIPLIB are tagged, we can save all the results of each batch of experiments in a CSV file, which will be merged with the known metadata about MIPLIB instances and then turned into a pandas data frame. We then can use an interactive dashboard, that can be created easily from the existing code using Streamlit [23], to filter the visualization by tag and spot instances of subcategories where our rules are particularly effective or not. For example, we expect to see worse results on set covering problems, caused by their entirely binary composition.

5.2.2. GPU acceleration

Since reordering is a computational effort that can take several minutes in the case of larger problems, it may be interesting to explore methods to speed up computations. Most of the operations required during the reordering are matrix and vector multiplications, that on paper should be faster if executed in parallel on specialized hardware such as GPUs. However, we should remember that most of the entries in our matrices are zeros and this poses a significant challenge for GPUs, which are architected to perform well on computational tasks with high arithmetic intensity and uniform memory access patterns, which happens in dense matrices or image processing tasks. The non-uniformity of memory access patterns can degrade the throughput of GPU cores. Also there is additional overhead when it comes to transferring data between CPU and GPU memory. Additionally, the steps that the solver executes, which include some presolve and postsolve logic, are often responsible for structure changes of the underlying problem which requires several reloads in memory that further degrades the effectiveness of parallel accelerators. Finally, many algorithms within the solvers heavily rely on caching, which is available on the CPU but may not map well on GPU-based hardware.

For all of these reasons, we will not use specialized hardware but instead leverage the availability of multiple cores in our testing environment. Solvers like Gurobi are exploiting multi-cores and multi-threading, but in the presence of a very high number of cores, we will launch multiple experiments in parallel, with the objective of fully exploiting CPU resources on our machine.

5.3. Permutation granularity

Now we should spend some more words on how many ways we have to permute a matrix. We defined what mathematically a permutation means in matrix form, but what we did not consider is that we are not forced to permute exactly every row and column, we could for example choose to have a granularity of more elements, and what we are going to permute will be subsets of more elements. This is because what we are aiming to mitigate with our reordering algorithm is the adversarial attitude of a client or user, who is trying to perturbate a problem instance in order to make our solver worse in terms of solving time stability. But it is unlikely that these perturbations will look like completely random problems, especially in the case that the initial problem formulation has a recognizable structure. Figures 5.1 and 5.2 show one example of a permutation of a problem with granularity one, which means that every row and column will be permuted. From now on, we will refer to this as an "all" permutation. Whenever we mention very random or highly random structures, it is because we are using a granularity setting that is very close to an "all" permutation.

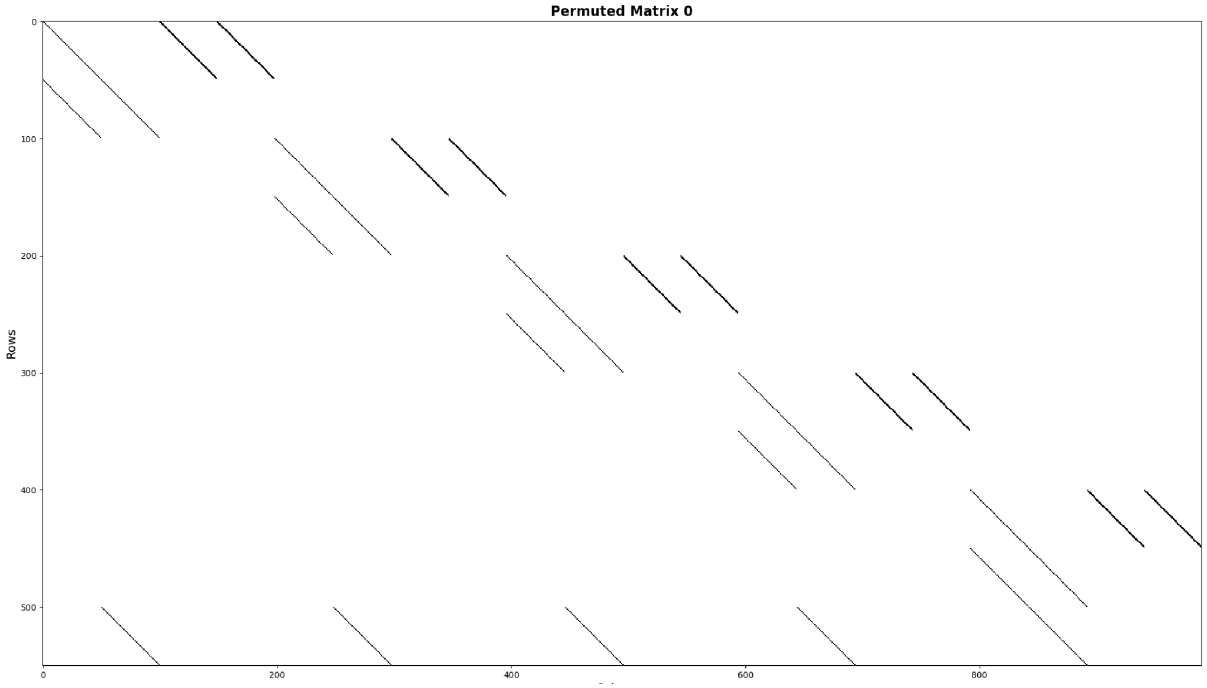

Figure 5.1: Original structure of the `exp-1-500-5-5` matrix instance before any permutation or rule application.

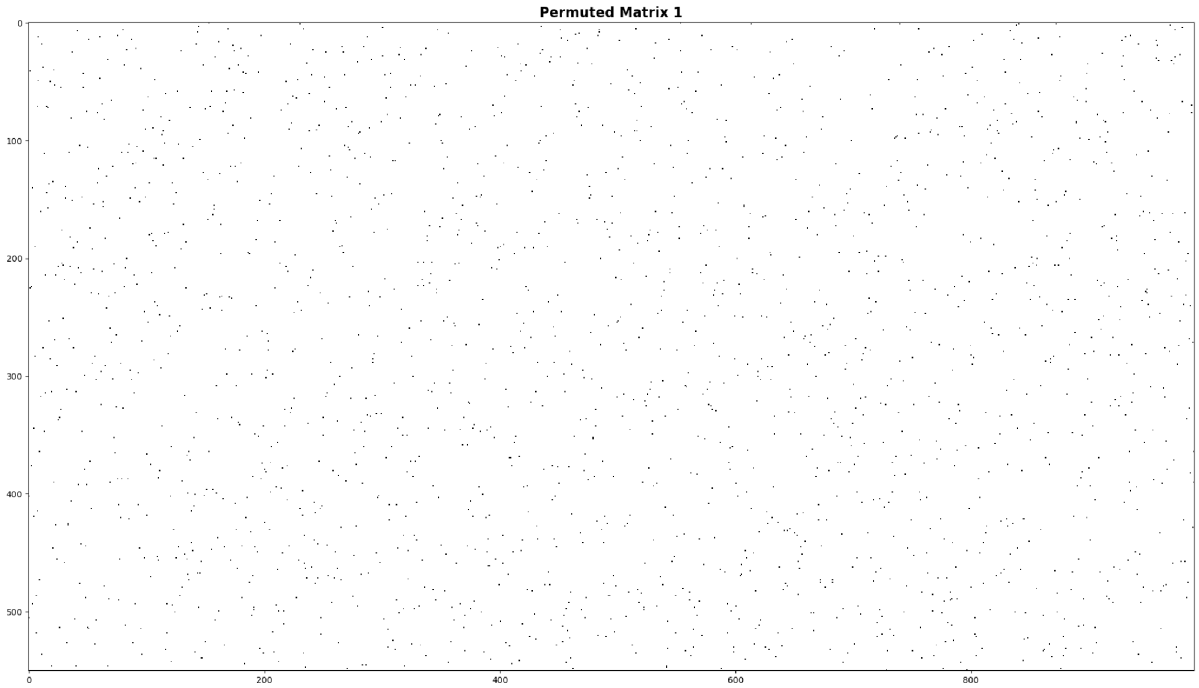

Figure 5.2: Randomly permuted version of the matrix `exp-1-500-5-5`, with the granularity set to every individual row and column. This configuration fully scrambles the structure, concealing any visual modularity and making any original block structure unrecognizable

As we can see the original problem had some kind of structure, which is completely lost after the permutation in the right image, where we can only see a cloud of random points.

What we should ask ourselves now is whether that structure in the original problem is common to find in instances of the MIPLIB. The answer is that it is a feature which is quite diffuse, particularly in every problem instance that models a real-world scenario, and is caused by how mathematical models are built. There are conventions that impose a certain order between types of constraints, for example, which results in a model that often exhibits a sparse structure, that is similar to a block diagonal at first sight. Also, we have to consider that solvers are tuned to perform better on problems which have these ordered structures in their formulation. In addition, there are often some presolving steps that have the effect of making the formulation converge even more to such regular shapes.

Considered all these reasons, we should try to set up our experiments so that the solving times are not influenced by those structural differences. This includes strategies where we decide to use a coarser granularity, such as in Figure 5.3, to maintain the preferred structure by solvers or at least do not mix very randomized permutations with too regular ones, as we did in the example before.

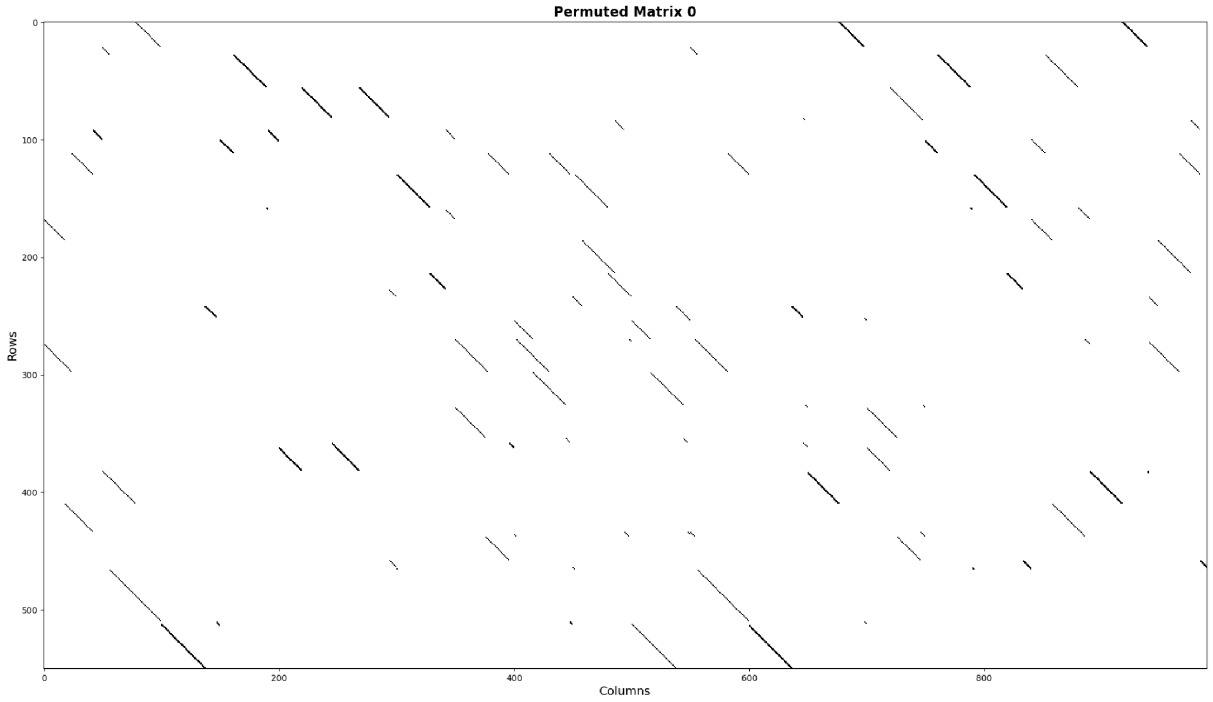

Figure 5.3: Permutation of the `exp-1-500-5-5` matrix using a block-level granularity of 20. Rows and columns are permuted within each of the 20 equally sized blocks, preserving local structure but disrupting global patterns. This intermediate level of permutation retains some visible clustering while partially obscuring the original modular layout

In mathematical terms this is done considering a block-wise permutation instead of permuting every row and column individually. Let n be the number of variables and m the number of constraints in a mixed-integer linear program. Given an integer k such that $k \mid n$ and $k \mid m$, we partition the set of variable indices $N = \{1, \ldots, n\}$ into k disjoint blocks $B_1, \ldots, B_k$ of size $\frac{n}{k}$, and the set of constraint indices $M = \{1, \ldots, m\}$ into blocks $C_1, \ldots, C_k$ of size $\frac{m}{k}$. We then define block permutation matrices $P_{\text{col}}^{(k)}$ and $P_{\text{row}}^{(k)}$, which permute the variable and constraint blocks according to a permutation $\sigma \in S_k$.

Let $\tilde{\mathbf{x}} = P_{\text{col}}^{(k)}\mathbf{x}$ denote the permuted variable vector. Applying these permutations, the transformed problem becomes:

$$\begin{aligned} \min \quad & \left((P_{\text{col}}^{(k)})^T \mathbf{c}\right)^T \tilde{\mathbf{x}}, \\ \text{s.t.} \quad & \left(P_{\text{row}}^{(k)} A (P_{\text{col}}^{(k)})^T\right) \tilde{\mathbf{x}} \leq P_{\text{row}}^{(k)} \mathbf{b}, \\ & \tilde{\mathbf{x}}_j \in \mathbb{Z}, \quad \forall j \in J' := \{\pi(i) \mid i \in J\}, \end{aligned}$$

where $\pi \in S_n$ is the permutation of variable indices induced by $P_{\text{col}}^{(k)}$. The integrality

constraints are updated accordingly to reflect the new variable ordering, as detailed in Chapter 1.

5.4. Pairwise distances

We have already defined Kendall Tau as the measure of distance we will use throughout our experiments, but we did not consider that we do not have a reference point from which we can measure distances. When dealing with matrices, we would expect to have a reference reordered matrix to which all the other reordered matrices will be compared to obtain our distance. Then to assess the effectiveness of our rules we could consider that smaller distances correspond to better rules. However, this is not feasible in our scenario as we do not have a reference. In fact, we should remember that our goal is to only decrease variability, not reordering the matrices to one specific form that is characterized by best performances for example.

That is why we are going to compare all the reorderings pairwise. This way, even if we do not have a reference form we can still determine how much the distance varies and if the various matrices have a similar form. This approach is feasible only in the case we have a low number of matrices. Luckily, as we said earlier, we do not need a big number of permutations, in fact, four permutations of the same problem are enough to assess its variability. In mathematical terms, this will result in computing all the possible pairs between permutations and calculating the variability between them as specified in an earlier chapter. For completeness, we report here the math behind this reasoning.

Given a set of n permutations, the pairwise distance between two permutations π_i and π_j is defined as:

$$d_{ij} = \text{distance}(\pi_i, \pi_j)$$

where d_{ij} is the computed distance metric between the two permutations.

The standard deviation of the set of pairwise distances is given by:

$$\sigma = \sqrt{\frac{1}{m-1}\sum_{k=1}^{m}(d_k - \bar{d})^2},$$

where:

- $m = \binom{n}{2} = \frac{n(n-1)}{2}$ is the number of unique pairs.
- d_k represents the individual pairwise distances.

- $\bar{d}$ is the mean pairwise distance, computed as:

$$\bar{d} = \frac{1}{m} \sum_{k=1}^{m} d_k.$$

6 | Runtime Behavior and Evaluation of Reordering Techniques

With our test environment and our list of benchmark instances from the MIPLIB, we were able to run the experiments, gather results, and try to assess the effectiveness of the ordering algorithm along with the various rules that were part of it.

6.1. Hierarchical rules results

Assessing the results of hierarchical rules made sense even after all the considerations we had made earlier about scaling. That was because, for the sake of our experiments, we started by considering only problem permutations and understanding how much reordering a problem could impact variability. In addition, we also had a term of comparison when we assessed the recursive rules, which although might have generalized better, could have had worse performances in very specific cases.

The set of rules used during this experiment were:

1. VariableTypeRule (8.1.1)
2. BoundCategoryRule (8.1.2)
3. ColumnsCoefficientRule (8.2.1)
4. ObjectiveCoefficientRule (8.2.2)
5. VariableOccurrenceRule (8.2.3)
6. ConstraintSenseRule (8.2.4)
7. ConstraintCompositionRule (8.1.3)
8. RowCoefficientRule (8.2.5)

9. RHSValueRule (8.2.6)
10. ConstraintRangeRule (8.2.7)

We started by looking at permutation distances between permutations and their reordered forms.

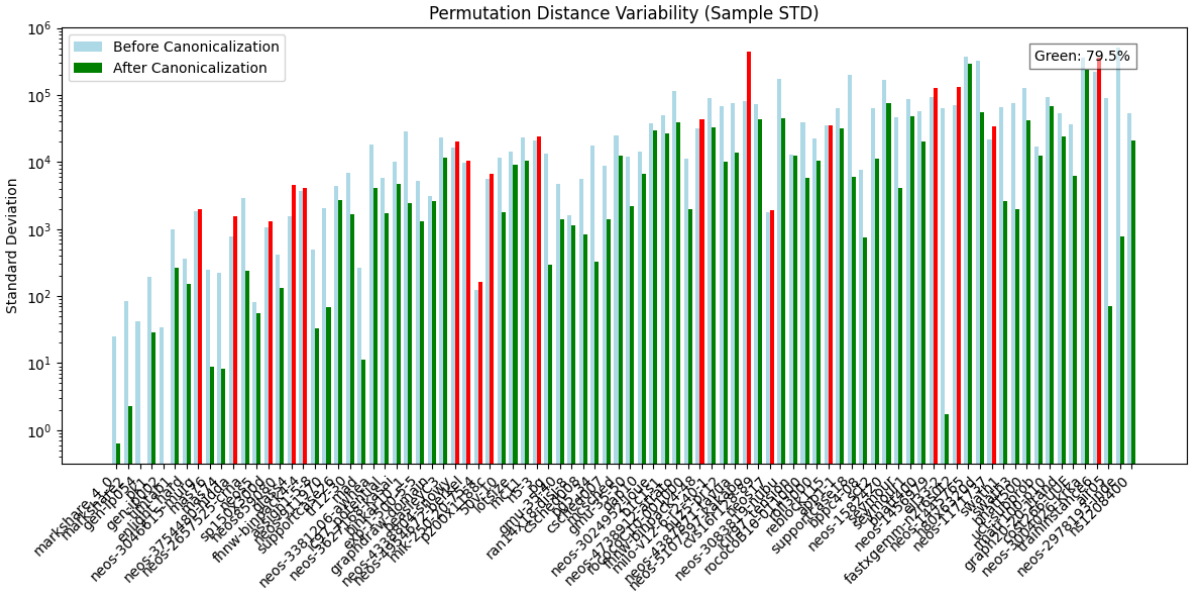

Figure 6.1: Permutation distance before and after applying hierarchical reordering rules. For each instance, we show two values: the distance between different random permutations (left bar) and the distance between the corresponding reorderings (right bar). Lower distances on the right indicate that the rules consistently produce more similar structures. The percentage shown in the top-right corner indicates the proportion of instances where the reordering distance is smaller than the original permutation distance

What we see from the graph in Figure 6.1 was a comparison of how permutation variability changed before and after applying our reordering. For the sake of simplicity, we referred to the reordered form as the "canonical form" from that point onward and the process of obtaining it through a reordering as "canonicalization". It was clear that applying our heuristic had an impact on permutation distance variability in the vast majority of instances, in particular a reduction that affected about 80% of the instances. Note that the y-axis has a logarithmic scale, which made it easier to notice the effect.
Even though in some cases the reduction seemed very slight, in others there was a clear difference of more than one order of magnitude. This was expected as our heuristic did not aim to be universally applicable but rather to have a great impact on some problem categories.

Although the previous metrics gave us some information, we were more interested in a metric that more intuitively explained how much permutation distance had decreased in the average case.

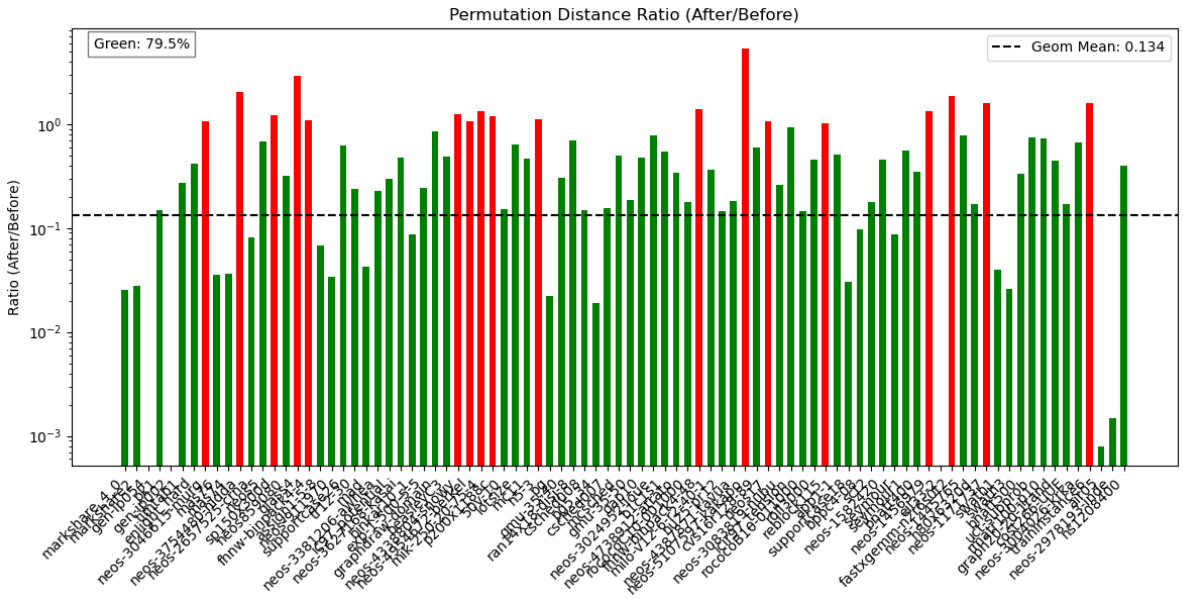

Figure 6.2: Permutation distance ratio with hierarchical rules. For each instance, we report the ratio between the variability of the reordered forms and the variability of the permuted forms. A ratio lower than one indicates that the hierarchical rules reduce structural variability across permutations. The top-left label shows the percentage of instances with a ratio below one, while the top-right label reports the geometric mean of all ratios

That was why we introduced the ratio represented in Figure 6.2. Dividing the value after canonicalization by the value before it, we always obtained a value between zero and one that was more representative of the reduction. Also, taking the geometric mean gave the right importance to terms when averaging, in contrast to a simple mean that would have given too much importance to outliers and classes of problems where our heuristic was not effective. The mean value we obtained clearly stated that the reduction in permutation distance variability was very relevant, reaching a value which was less than 15% compared to before applying the rules.
Even though these results already seemed very promising, we had to remember what we had discussed earlier about scaling and its impact on these rules we had just tested.

6.2. Recursive rules results

When considering recursive rules, we had to take into account that these rules were composed in a recursive way and gave different scores based on the number of terms inside submatrices where they were applied. This meant that the order of application of the rules mattered. First of all, we needed to find a method to determine the most appropriate order of application. That time we had more metrics to consider when assessing results. As we had stated in an earlier chapter, we reasonably thought there was a correlation between fine granularity in matrix decomposition by our rules and the effectiveness of the reordering. We just carried out different trials where rules were swapped, and we took the order that produced the finer granularity. Note that this approach was only valid for rules based on counting, while other rules could be applied at the beginning to distinguish the first subproblems within an instance, such as rules based on variable type subdivisions.

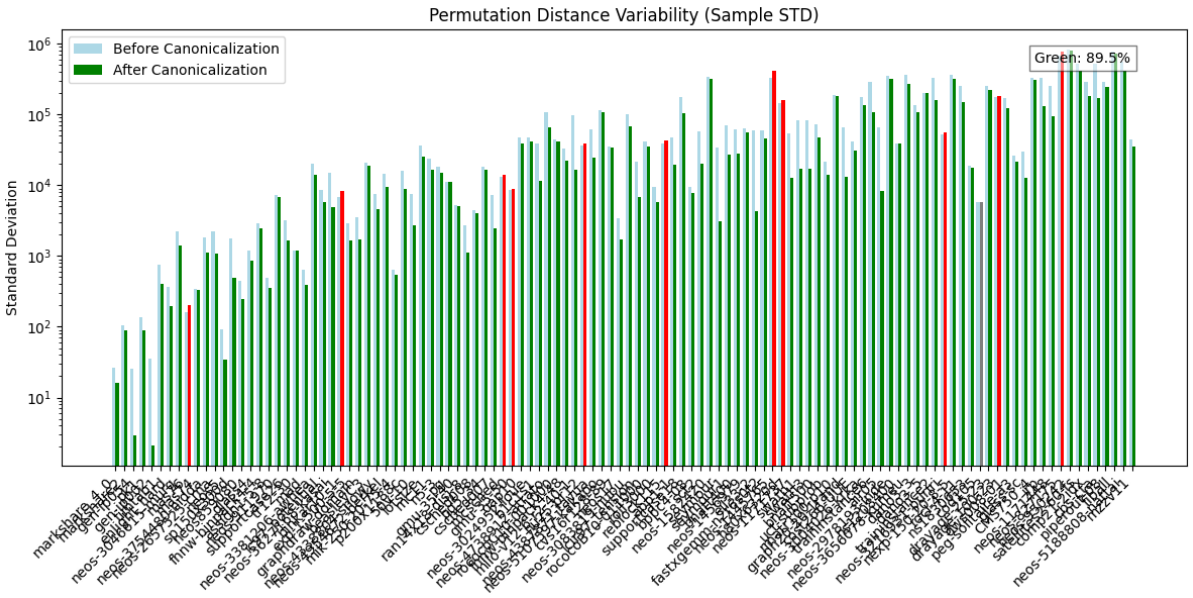

Figure 6.3: Permutation distance before and after recursive rules application. For each instance, we show two values: the distance between different permutations (left bar) and the distance between the corresponding reorderings produced by the recursive rules (right bar). Lower distances between reorderings indicate that the recursive rules yield more consistent structural forms across permutations. The top-right label reports the percentage of instances where the distance after reordering is lower than before

The set of rules used during this experiment were:

1. VariableTypeRule (8.1.1)
2. BoundCategoryRule (8.1.2)

3. ConstraintCompositionRule (8.1.3)
4. AllBinaryVariablesRule (8.3.1)
5. AllCoefficientsOneRule (8.3.2)
6. NonZeroCountRule (8.3.3)
7. SignPatternRule (8.3.4)
8. ConstraintIntegerCountRule (8.3.5)
9. ConstraintContinuousCountRule (8.3.6)
10. BothBoundsFiniteCountRule (8.3.7)
11. BothBoundsInfiniteCountRule (8.3.8)
12. OneBoundFiniteCountRule (8.3.9)

After this premise, we started looking at the results with the same graphs we had seen in the hierarchical case.

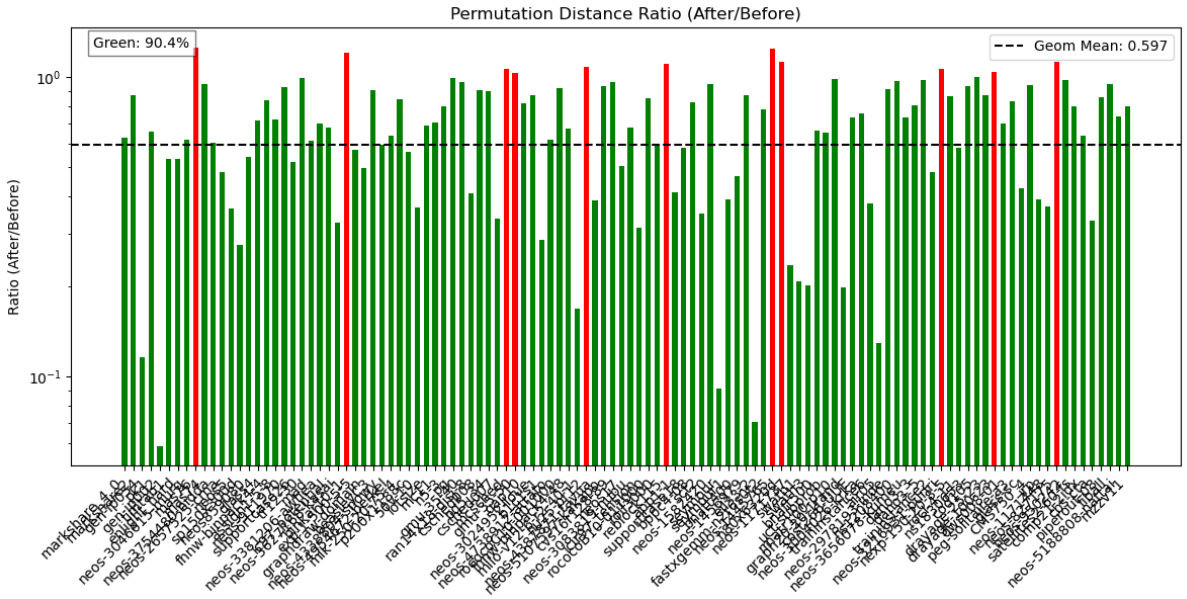

Figure 6.4: Permutation distance ratio with recursive rules. For each instance, we report the ratio between the variability of the reordered forms and the variability of the original permutations. A ratio below one indicates that the recursive rules reduce structural variability, producing more consistent forms across permutations. The top-left label shows the percentage of instances with ratio below one, while the top-right label reports the geometric mean of all ratios

It was clear from the data in Figure 6.3 that we achieved a decrease in permutation distance in more cases, reaching roughly 90%. It also seemed we had more stability, as in the cases where permutation distance was not reduced it was actually very close to the beginning, meaning that at least the perturbations introduced were somehow more mitigated than before.

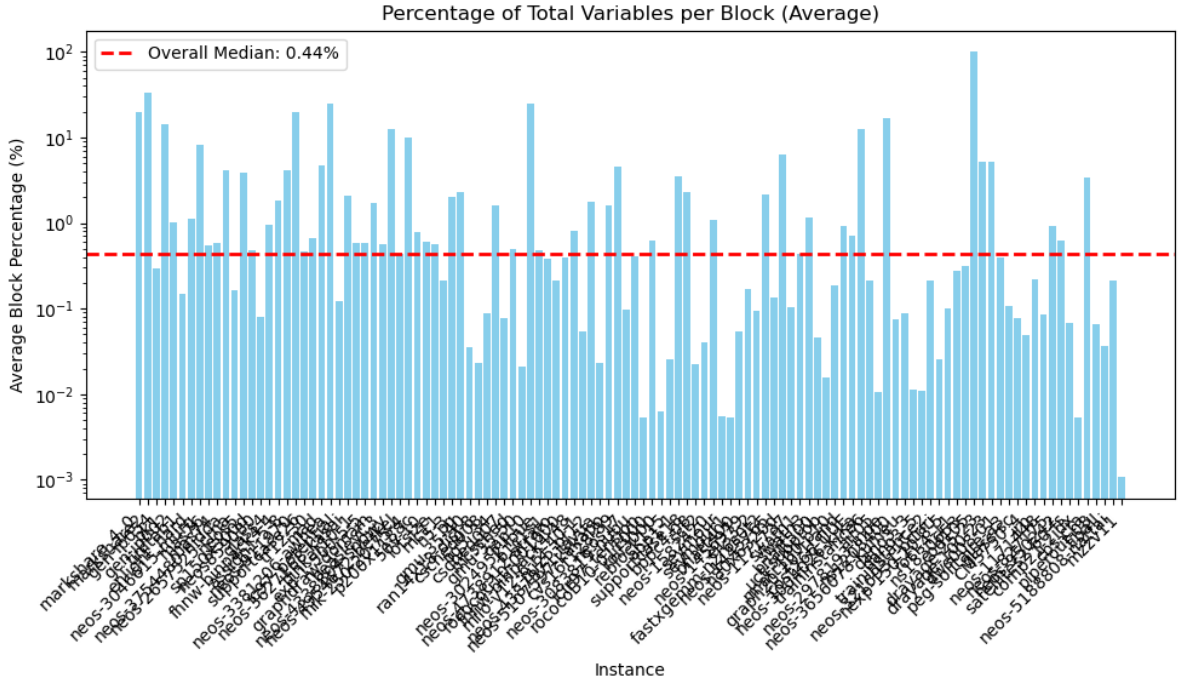

Figure 6.5: Block size median compared to original matrix size with recursive rules. For each instance, we report the median block size found by the algorithm, normalized by the original matrix size. Lower values indicate finer granularity, meaning the recursive rules identified more detailed structural patterns. The top-left label shows the overall median of medians across all blocks

Unfortunately, when we looked at the mean of the ratios in Figure 6.4 we clearly saw that the performance was worse compared to the hierarchical rules. This meant that this heuristic worked better to generalize across different types of instances, but it did not produce a big reduction as we had seen before with the hierarchical approach. Nevertheless, we had a median of roughly 60%, meaning that there was still a significant reduction.

We then considered granularity. To have a metric that encompassed the information we needed to understand if our rules produced sufficient granularity, we considered the median block size compared to the whole matrix size, calculating what percentage of variables was within a block on average on each instance. We then took the median value across instances to avoid being impacted by outliers, which were those cases where our

rules were not finding blocks when applying the algorithm.

As we saw from the graph in Figure 6.5, we were obtaining quite satisfying results, as our median value was less than 1%.

6.3. Low performance subset

When looking at the granularity graph, we saw that there were some problems which had way higher values, and interestingly there was even one problem where granularity was 100%, meaning that none of the rules was able to find a subdivision inside its matrix formulation. We expected that these problems had some common features and, if we filtered our results by labels that were assigned to each MIPLIB instance as part of its metadata, we could find that all these problems were part of a particular class. In order to do that, we inspected the names of the instances and tried to find an intersection between the labels they had assigned to them.

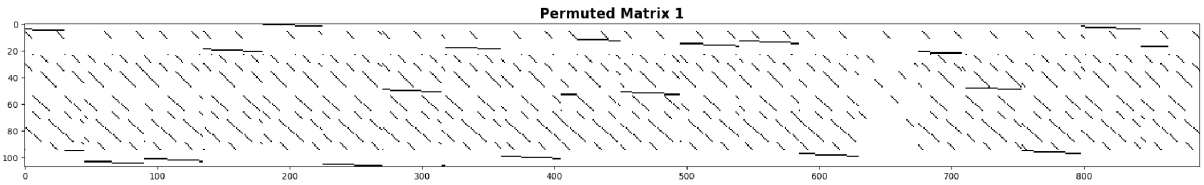

Figure 6.6: Permutation of the `neos-911970` instance with block granularity set to 20. Rows and columns are shuffled within 20 equally sized partitions. This disrupts the natural structure of the matrix, obscuring any underlying block or cluster patterns.

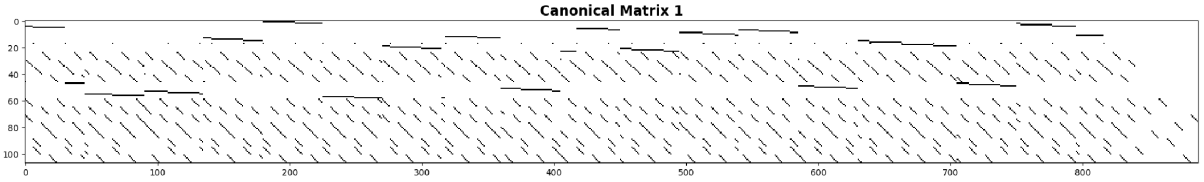

Figure 6.7: Reordered version of `neos-911970` using only the recursive reordering rules. The result fails to produce a clearly interpretable block structure, indicating that the recursive strategy alone is not sufficient for this instance

As we had predicted earlier, it turned out that these instances were the ones that had only binary variables, such as set packing, set covering, and set partitioning problems. These problems had structures that were indistinguishable by the current rules and might have needed some specialized ones. While their objectives had different forms and coefficients, they had in common very intricate combinatorial structures and very sparse formulations, with a majority of decision variables taking a value of zero, along with many structural

symmetries and regularities making variables and constraints indistinguishable when they were randomly permuted. This suggested to us that we needed rules that were more sensitive to other features, needing a different approach than the counting rules of non-zero entries that we had applied so far.

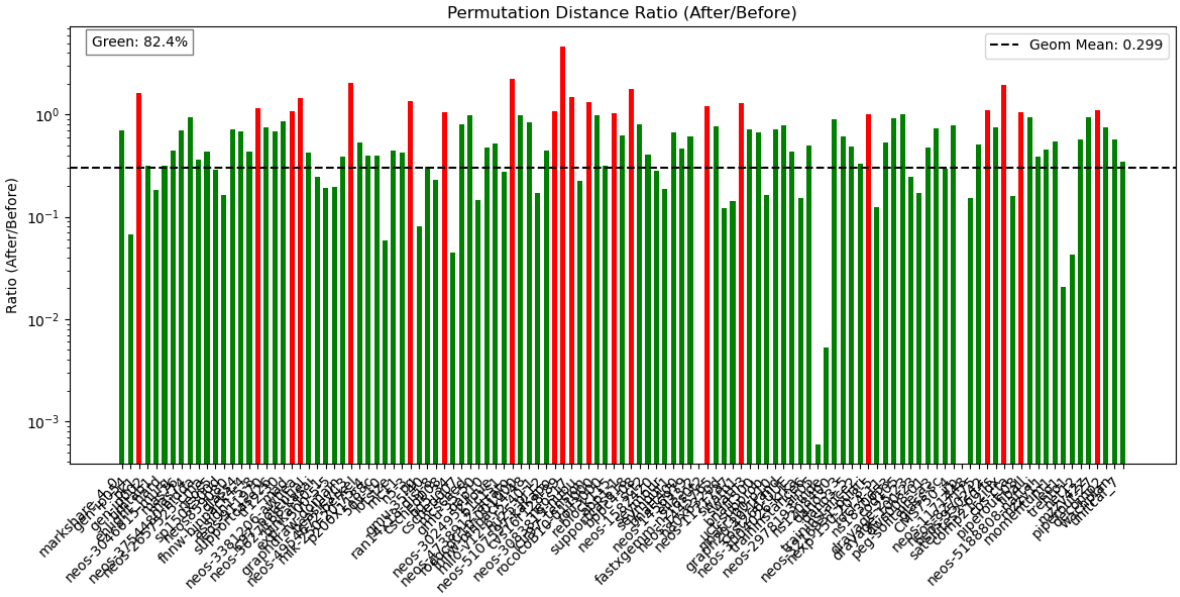

Figure 6.8: Permutation distance ratio after applying recursive rules followed by binary-structure-aware rules. The ratio compares the variability of the reordered forms to the variability of the original permutations. Values below one indicate that the specialized rules further reduce variability, producing more consistent reorderings. The percentage of instances with reduced variability is shown at the top left, and the geometric mean of the ratios is displayed at the top right

We can present a representative example where the recursive reordering strategy alone performs poorly, and where the introduction of additional problem-specific rules significantly improves the structure of the reordered matrix. The instance `neos-911970` showed in Figures 6.6 and 6.7, a set covering problem, is used to demonstrate the comparative impact of the applied rules.

Since, as we had seen earlier, the granularity we obtained on average was already satisfying, we were not expecting to have way better performances just by reordering the existing rules. Instead, the approach we needed in that case was to come up with a specialized set of rules that would be applied at the end after all the others, and would be effective on those submatrices that were still very big, as they had not been divided by the previous rules. Note that this set of rules took for granted that all the others had already been applied. This meant they were not designed to be applied at the beginning, even if we

wanted to explore the results of different orderings of rules, as they would have disrupted the performance of the whole algorithm.

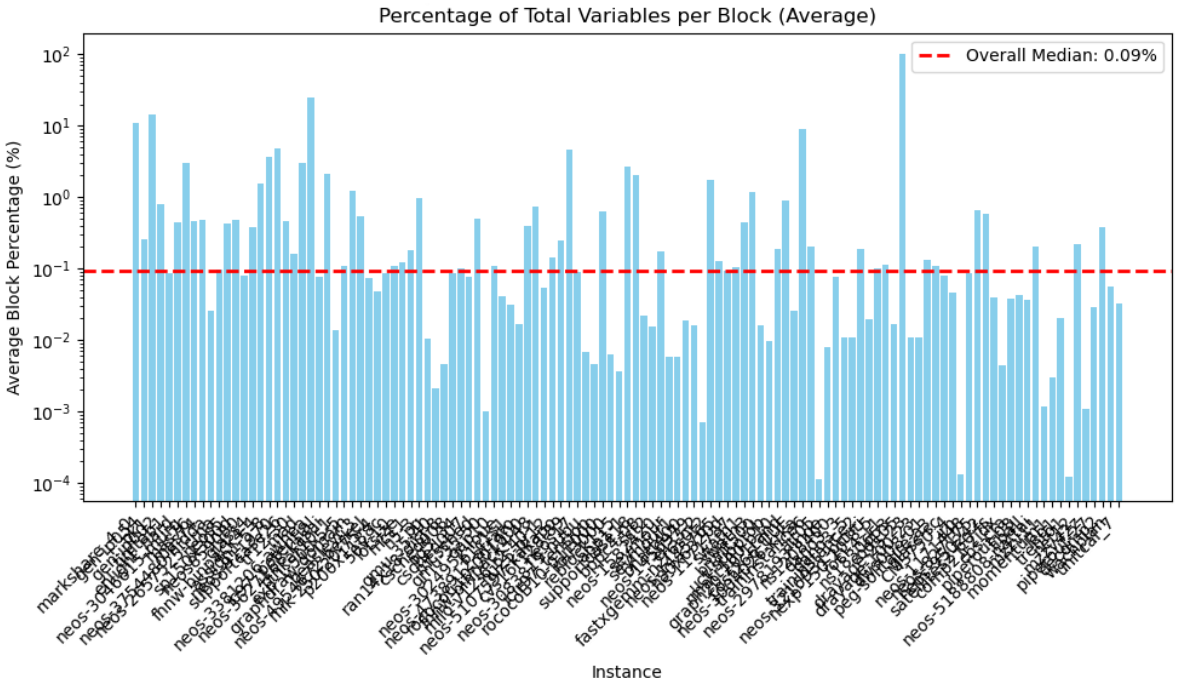

Figure 6.9: Block size median compared to original matrix size with recursive rules and additional specific rules. For each instance, we report the median block size after applying the full set of rules. Lower values indicate finer granularity and more effective structure detection enhanced by the specialized rules. The top-right label shows the overall median of medians across all blocks

With a guarantee that all the variables were binaries, which was double-checked by the rule itself, we looked at other features that, in the beginning, we had defined as not scale invariant, but became safe to measure in this case, as binary variables could not be scaled in ways that would have caused them to lose their binary nature.

The rules that detected these features and that were appended to the existing recursive rules were the following:

1. SetPackingRHSRule (8.4.1)
2. UnscaledObjectiveOrderingRule (8.4.2)

After applying these rules, we witnessed a significant improvement in our graphs showed in Figures 6.8 and 6.9 for all instances.

The mean permutation distance ratio had improved significantly, and in particular granularity had further decreased, confirming that all the matrices that had not been subdivided

efficiently by the previous rules had structural properties that we were detecting with the new specialized rules. If we looked at how the specific case cited above had changed, we saw in Figure 6.10 that a greater section had a recognizable order:

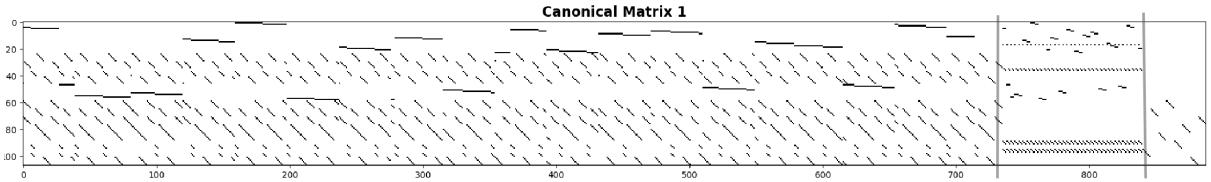

Figure 6.10: Reordering of `neos-911970` after applying additional problem-specific rules on top of the recursive strategy. This enhanced rule set reveals a much clearer modular structure, suggesting that domain-specific guidance is essential for effective reordering in some cases

6.4. Runtime variability

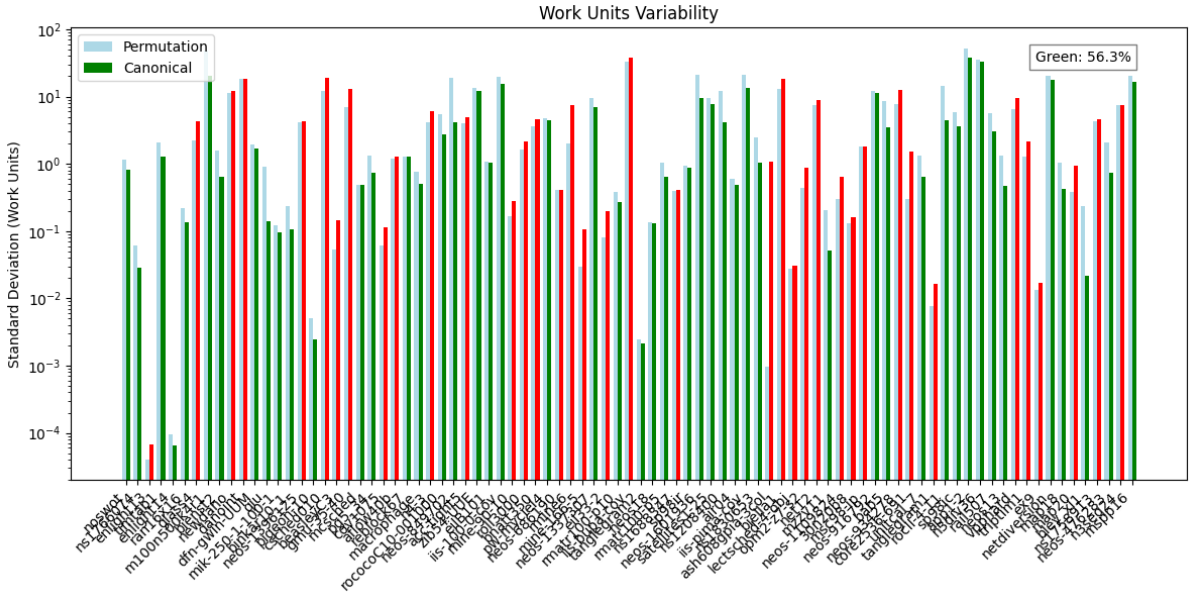

Figure 6.11: Work units variability before and after recursive rules application. For each instance, we report two values: the variability of solver work units across different permutations (left bar), and the variability after applying recursive rules (right bar). Lower variability after reordering indicates that the rules lead to more stable solver performance across permutations. The percentage of instances where variability decreases is shown at the top right.

After having found and refined our set of rules which was robust to scaling and edge cases in problem structures, it was time to look at solve time to understand if the results

matched what we had found in terms of permutation distance. As we stated earlier, we considered work units instead of solve time in seconds, since they expressed a measure of runtime which was less influenced by the specific hardware and parallelism used during the solving process.

In our experiment results, we had graphs structured similarly to before in the case of distances and showed in Figures 6.11 and 6.12.

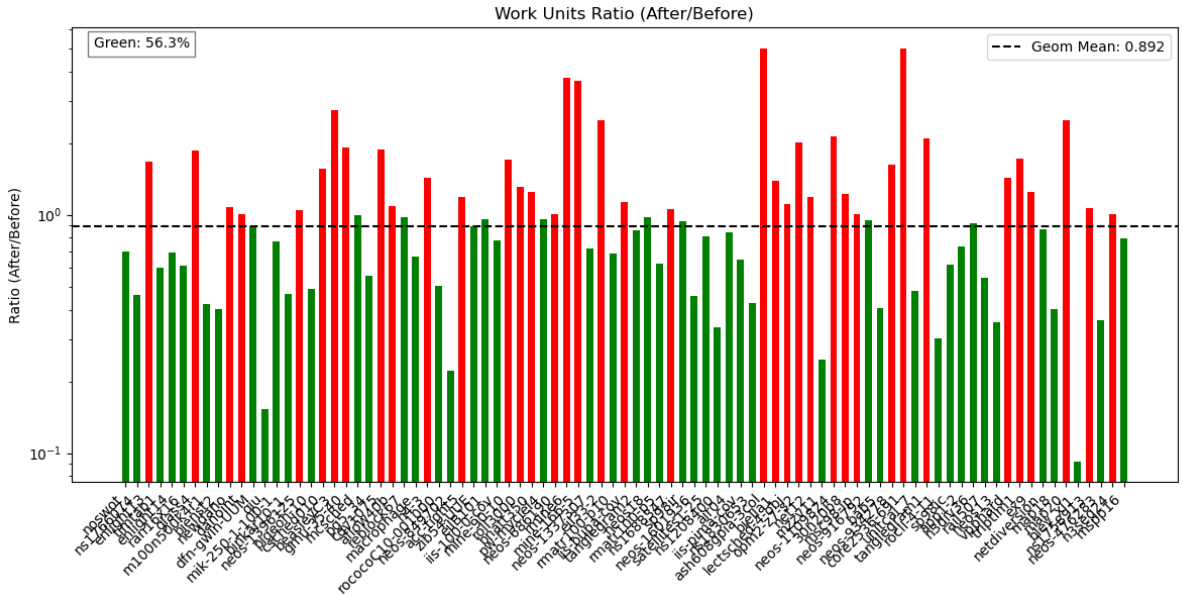

Figure 6.12: Work units ratio with recursive rules. The ratio compares the variability in solver work units after applying recursive rules to the variability across permutations. A ratio below one indicates that recursive reordering leads to more consistent and often reduced solver effort. The percentage of instances with reduced variability is shown at the top left, and the geometric mean of the ratios is shown at the top right

What we noticed immediately was that we did not have matching patterns with permutation distance results. The first graph clearly showed us that the reordering had made our solve time lower only slightly more than 50% of the times, which was not a satisfying result since we knew from the literature that randomly permuting a problem had an equal probability of making it faster and slower to solve [14]. This implied that on a large instance set we expected to have half solve times higher and half solve times lower. If we looked at the geometric mean of ratios, we saw that we were only a few decimals below one. This meant we were not significantly reducing variability, but at least our algorithm went in the right direction actually decreasing solve time on average.

6.5. Different permutation granularity

As we mentioned earlier, we were not interested in permuting necessarily all rows and columns, in fact, we considered the impact of having permutations of different granularities and aimed to find which ones were mitigated more by our rules. In the experiment before, we had set granularity to one variable per block in order to simulate the most adversarial behaviour of an actor that tried to disrupt the conditions preferred by solvers, whose algorithms were often fine-tuned on original problem structures of the MIPLIB, which presented clear patterns. When dealing with real-world problem formulations, we were interested in mitigating similar conditions, but it was more likely that the permutations in the problem were caused by slightly different formulations that caused swaps of a small subset of rows and columns in the coefficient matrix. That is why we analyzed the results of the same experiment we discussed in the previous paragraph but this time repeated with different permutation granularities. We tried the experiment with 10, 30, 100, 300, 1000, 3000, and "all" blocks. "All" represented the case where each variable had its own block so every variable was permuted, on the other hand, when we chose 10 blocks we had $n/10$ variables per block and $m/10$ constraints per block, where m and n were the number of rows and columns in our coefficient matrix A.

In Figure 6.13 we observed the resulting geometric means of ratios for the different granularity choices.

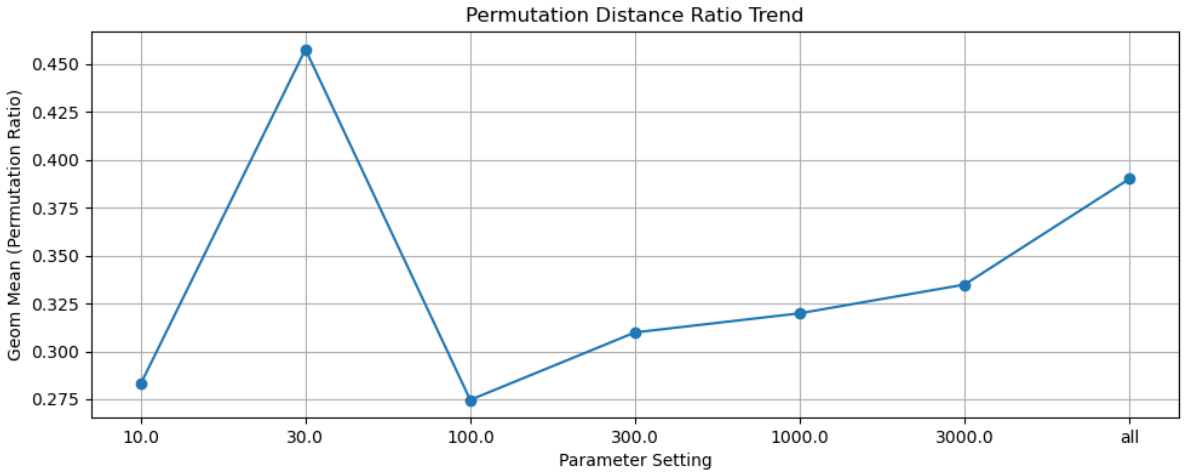

Figure 6.13: Permutation distance ratio geometric mean for different permutation granularities. Each point represents the geometric mean of the permutation distance ratios across all instances for a given granularity level. Lower values indicate that the rules reduce variability more effectively

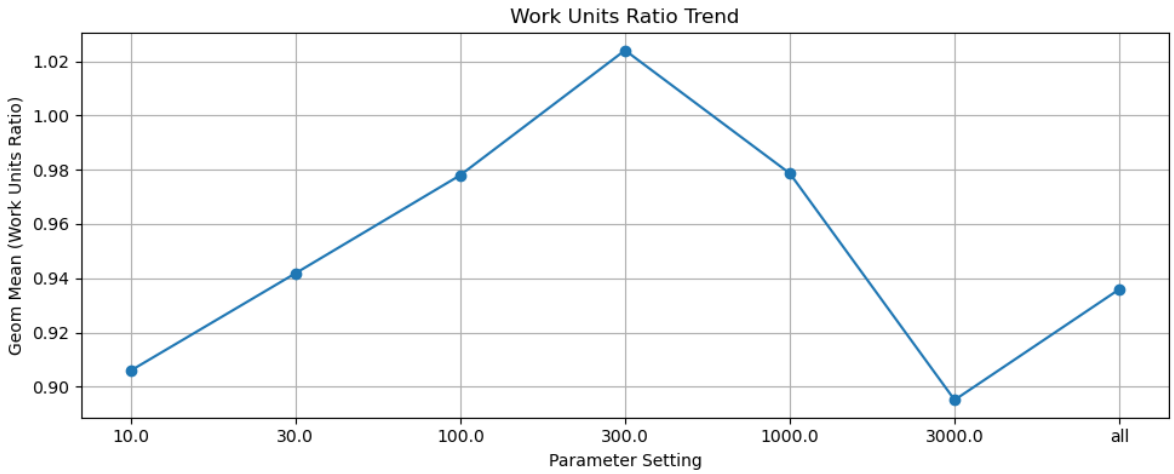

Figure 6.14: Geometric mean of work units ratio for different permutation granularities. Each point corresponds to the average ratio of work units before and after applying the rules for a specific granularity level. Values close to one indicate minimal impact on runtime variability, suggesting that the permutation granularity has little effect on the reduction of work units by the rules

As we could see, the graphs did not seem to have a clear trend. We concluded that for a few blocks, which had a lot of variables and constraints inside, the disruption in permutation distance could be uneven in the matrix, causing some oscillations. For a greater number of blocks, it made sense that having a finer granularity led to an increase in permutation distance since the permutations were more random.

If we looked at work unit results in Figure 6.14, we found an unexpected pattern which seemed not to give us any information or apparently told us that the 300 blocks was the worst choice. If we looked more closely at the range of results in the y-axis, we saw that actually, the results were very close between them and also very close to one, so basically we expected to have an oscillating behaviour between values slightly below one, most likely caused by a small numerical instability within the solver. This meant that neither of these permutation granularities was preferred by our rules and had a very limited impact on reducing variability.

7 | Automatic Decomposition and Results Consolidation

Even though our rules-based approach has an impact on permutation distance and it is clearly capable of detecting structures within matrices, as the problems get bigger and have less recognizable patterns, it starts to become less practical. Furthermore, from the results of the experiments, we observed that it does not have a significant effect in the mitigation of solve time variability. These facts suggest to us that it may be worth considering other ways to find structures within problem matrices.

7.1. Automatic Decomposition

In literature we can find automatic decomposition techniques that are used by state-of-the-art solvers. Some implementations of decomposition techniques such as GCG [8] or DIP [7] tackle large-size instances, where trying to solve the whole problem is inefficient and a subdivision into subproblems is applied to better exploit similarities of different parts of the matrices. These decomposition techniques are a preliminary step to apply well-known methods like Benders and Dantzig-Wolfe which have both been discovered in the early 1960s and commonly used nowadays to solve large-scale optimization problems. The idea behind these methods is to find subproblems that are strictly interconnected, which can lead to smaller solve times and more parallelism during the solving process. In our case we are not interested in increasing performances, but in stabilizing them. Nevertheless, we could still leverage these decomposition methods. That is because these block decompositions have the effect of grouping variables based on the similarity of certain features, such as complexity, to be interpreted not as the general concept but as what solver algorithms find harder to solve based on the algorithms they use.

The main reason why our previous efforts of building a unique form starting from different permutations of the same matrices seemed to be promising according to our distance measure, but the effect on solve time variability did not reflect the expectations is that being closer in structure for two matrices is not sufficient by itself to lead to close performance

during their resolution.

We have already pointed out that significant performance variability can be observed even when a few variables and constraints are permuted. That is because not every variable and constraint in the matrix has the same relevance. These differences are clear when turning real-world scenarios into mathematical terms because we know which features the variables are representing, but get completely lost when we are looking at coefficient matrices and their permutations. That is why a mere minimization of permutation distance like the one we used so far is insufficient to reorder the problem effectively. Of course, it creates a reordering that goes in the right direction, and the finer the granularity we find with our algorithm the closer we are to having an exactly matching order.

7.2. Extending the Experimental Framework

Since these automatic decomposition techniques are tailored to create subproblems that are similar and are more homogeneous according to the solving algorithms preferences, they may be a better starting point.

In the 2017 version of the MIPLIB, we have more information about the problems compared to the previous versions like the 2010, and luckily for most of the instances, we have some files which represent the decomposition of the instance. We should mention that solvers like GCG find many decompositions of the same problem which maximize different metrics. The most used metric and the one that corresponds to the files we find on the MIPLIB is the max-white-area, which is based on finding the most concentrated variable decompositions that lead to the majority of zero coefficients in the matrix outside of the blocks. To integrate these decompositions into our environment we can directly download the decomposition of each problem from the MIPLIB, without the need to have another solver to compute it. This lets us stick only to Gurobi for our experiments.
The decomposed structures are provided in a special file format (.dec) that we can parse according to what we find in the DECOMP documentation [21]. Once the decomposition for each problem is loaded in our testing environment, we can create a rule to subdivide the matrix into blocks as the decomposition specifies, this time not needing to apply it recursively. We can instantly observe that even though the decomposition is far less fine in granularity, our coefficients are more concentrated in blocks. This leads to a greater decrease in permutation distance.

At this point, we should try to understand if an ordering within blocks or even the order of blocks has an impact on variability. As of now, our blocks are just ordered decreasingly by size. One observation we should make about the algorithms used by solvers in their

first steps is how some presolving and branching techniques, such as strong branching, work and the fact that often only a subset of the available variables considered at the beginning of the algorithm is explored completely. This means that these first variables have a big impact on how long it will take to solve the problem, which means that for us it is important that these first sets of variables are as similar as possible across reordered permutations. The reason behind that is that early branching decisions are crucial in determining the structure of the branch-and-bound tree.
To mitigate the risk of making poor early decisions, solvers often employ a technique known as strong branching. In strong branching, the solver tentatively branches on candidate variables by solving a few iterations of the LP relaxation for each choice, thereby estimating the impact of branching on each variable. This allows the solver to select the variable that is most promising in terms of pruning the search tree. Strong branching was first introduced in the context of branch-and-cut by Applegate et al. [1]. This helps us with our reorderings but still, we need a great overlap of variables across different reordering to expect we have comparable results in terms of runtime variability. One strategy to achieve a block ordering that goes in this direction is to use some of the rules we already defined but this time to score and reorder blocks. One way to leverage the rules is to prioritize blocks that are more structured and have tighter bounds, which are preferred by the solver. Then proceed with the remaining ones from the sparsest to the denser and also reduce ties based on block homogeneity based on variable types within blocks.
Note that this order of blocks is not necessarily the same as the one followed by the solver, as it also depends on the presolve steps, which can include problem simplifications and permutations, as described earlier in this thesis. This was also true for the previous orderings found with other rules, but this time it is more likely that these presolve steps are not disrupting our ordering because the block decomposition has been made with methods that are more compatible with the preferences of solving algorithms.

7.3. Final Results

We executed a new set of experiments with the updated approach and compared the results to those obtained previously. As already observed, it was not possible to overlook the fact that different permutation granularities could produce varying effects. Therefore, we repeated the same experiment settings across different levels of permutation granularity.

We divided this final batch of experiments into three categories. First, we examined the impact of decomposition alone, as produced by GCG, without applying any further reordering. Second, we reordered the blocks obtained from GCG. Lastly, we applied our

recursive rules within the blocks created by GCG.

In the first case, we aimed to isolate the impact of automatic decomposition on solve time variability. This allowed us to understand the magnitude of the effect and to establish a baseline for evaluating the improvements that followed. From the results in Figure 7.1, we observed a significant reduction in solve time variability for granularity 10, which is representative of coarse granularities. However, the impact remained limited for very fine granularities, which were representative of completely random permutations. It was also important to note that, when this method performed at its best, the reduction in variability exceeded that observed with all previously applied rules.

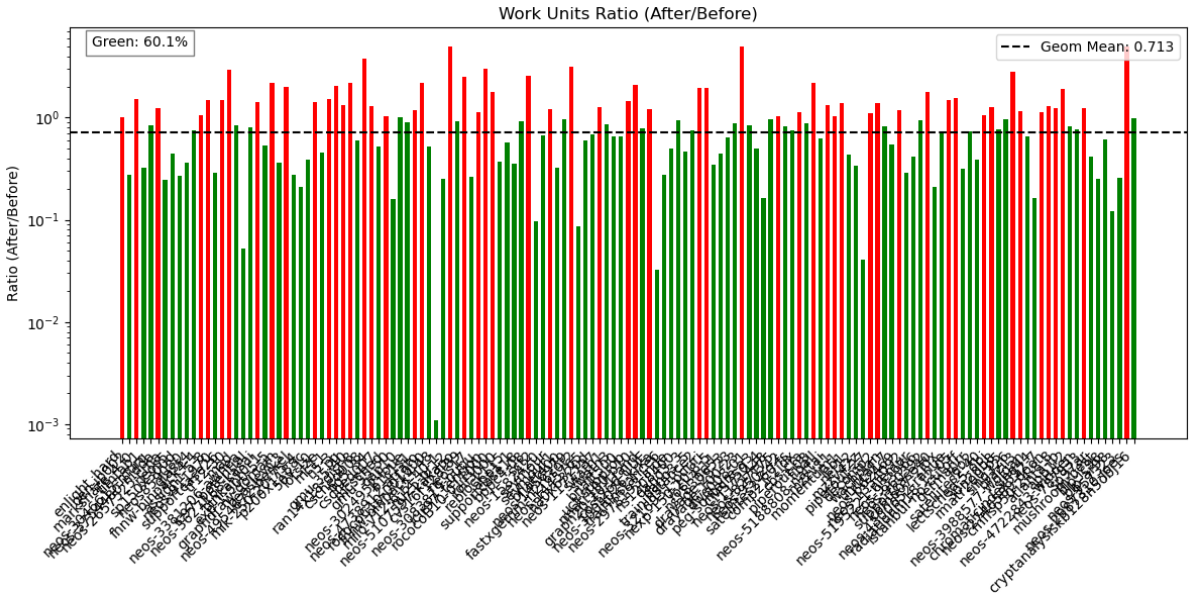

Figure 7.1: Work units ratio comparing original and reordered problems using only GCG decomposition rules. Values below one indicate a reduction in runtime variability due to the application of the decomposition, reflecting improved consistency in solver performance. The percentage of instances with reduced variability is shown at the top left, and the geometric mean of the ratios is shown at the top right

In the second experiment, we constructed a reordering of blocks that was more tailored to our objective. While GCG's output was intended for use with advanced solving strategies to enhance solver performance, our goal was to stabilize it. Therefore, reordering that prioritized smaller and denser submatrices could have led the solver to consider the same set of variables in the initial phases. This could have resulted in more stable outcomes, especially given our previous considerations about the early phases of solving and strong branching heuristics.

However, upon examining the results in Figure 7.2, this hypothesis did not hold. The ratios we observed were worse than in the first case, and the method still exhibited poor performance under highly randomized initial permutations.

Algorithm 7.1 Block Ordering Construction

1. `BlockSizeRule()` ▷ Prioritized smaller blocks
2. `BothBoundsFiniteCountRule()` ▷ Preferred blocks with more bounded variables
3. `NonZeroCountRule()` ▷ Preferred denser blocks
4. `ObjectiveNonZeroCountRule()` ▷ Preferred blocks with simpler objectives
5. `VariableTypeRule()` ▷ Preferred blocks with more homogeneous variable types

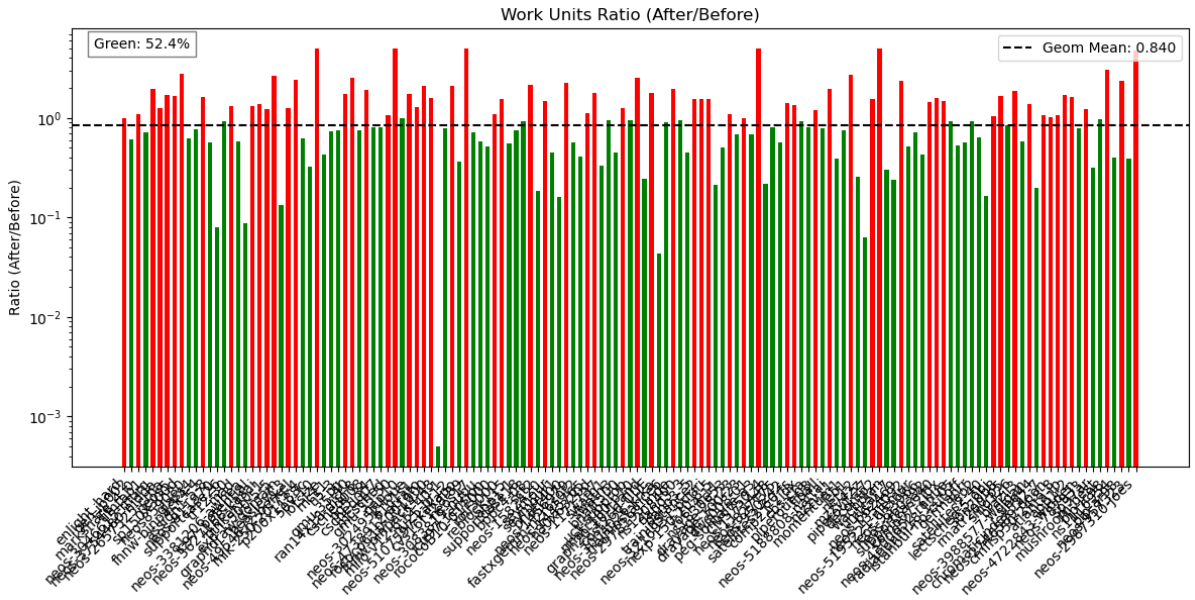

Figure 7.2: Work units ratio comparing original and reordered problems after applying both GCG decomposition and block ordering rules. Ratios below one indicate a reduction in runtime variability, suggesting that combining decomposition with block ordering improves the consistency of solver performance. The percentage of instances with reduced variability is shown at the top left, and the geometric mean of the ratios is shown at the top right

Finally, we combined block ordering with the recursive algorithm applied internally to the blocks. We used the same recursive rules described in the previous sections. This approach was motivated by the observation that GCG often produced a limited number of blocks, and that applying structure detection within those blocks could further stabilize solver performance.

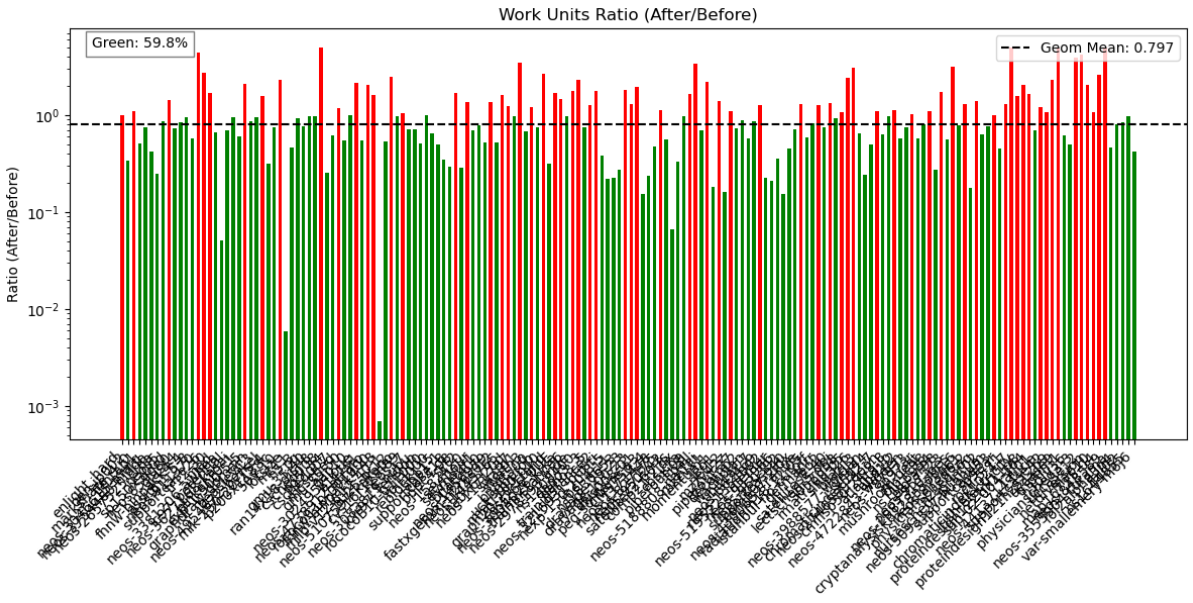

Figure 7.3: Work units ratio comparing original and reordered problems after applying GCG decomposition, block ordering, and in-block recursive rules. Ratios below one indicate that this combined approach further reduces runtime variability, demonstrating improved robustness and structure detection within blocks. The percentage of instances with reduced variability is shown at the top left, and the geometric mean of the ratios is shown at the top right

From the resulting graphs in Figure 7.3, we found that, similarly to previous cases, we did not observe a meaningful decrease in variability under highly randomized permutations. On the other hand, for coarser permutation granularities, the performance was essentially equivalent to the case without in-block reordering.

Table 7.1 summarizes the observed results across the different rule combinations and granularity levels.

In this last batch of experiments, we did not mention the results in terms of permutation distance. That is because we observed that every time GCG is used, which we can conclude is the main cause of solve time variability, our distance metric does not seem to show great results. We had already observed that there seemed not to be a great correlation between reduction in permutation distance and reduction in variability, and this last batch confirms this lack of correlation.

Rules Applied	Granularity 10	Granularity all
Recursive rules only	0.90	∼1.00
GCG only	0.70	∼1.00
GCG + block reorder	0.85	∼1.00
GCG + block reorder + in-block recursive rules	0.80	∼1.00

Table 7.1: Summary of solver time variability ratios (lower values indicate better reduction in runtime variability) obtained by applying different combinations of reordering rules at two permutation granularities: coarse granularity (10 blocks) and fine granularity (all variables permuted individually). The results show that GCG decomposition alone achieves the largest reduction in variability at coarse granularity, while adding block reordering and in-block recursive rules does not seem to provide improvements. At fine granularity, all approaches have little to no effect on variability reduction

7.4. Final Experiment Workflow

To execute the full experimental workflow, including the stage that leverages decomposition based on .dec files, we begin with the .mps instance files, which can be downloaded from MIPLIB. Once we have these, we start the framework and generate the desired number of problem permutations, specifying the permutation granularity.
For each permutation, we then run an automatic decomposition tool such as GCG, which generates the corresponding .dec file. The framework then finds the reordered form for each permutation, applying the rules specified for that experiment.
When using GCG's decompositions, the framework reads the .dec file for each instance, subdivides the problem matrix according to the defined blocks, applies block ordering (if enabled), and recursively subdivides each block into smaller blocks based on the active rules. In the end, we obtain a reordered form for each initial problem permutation. Next, we compute our evaluation metrics. For permutation distance, we calculate the standard deviation of the pairwise permutation distances between all problem matrices. We repeat the same calculation for the reordered forms, and then compute the permutation distance ratio as the ratio of these two standard deviations.
We follow a similar process for solve times. First, we solve each permutation and each reordered form using Gurobi. Then, we compute the solve time variability as the standard deviation of solve times across permutations, which we express in terms of work units, and again for the reorderings. The ratio of these two values gives us the solve time ratio.

Example with neos5

We revisit the neos5 problem from MIPLIB, which was previously used as an example in Chapter 1. In this section, we consider four distinct permutations, which are illustrated in Figure 7.4.

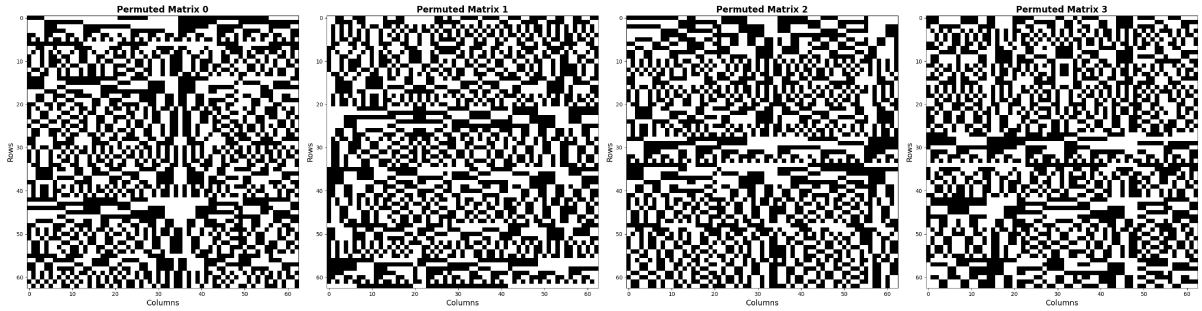

Figure 7.4: Permutations of neos5

We then apply the decomposition identified by GCG, which in this case finds four blocks, together with the block ordering and the recursive rules applied within each block. The resulting reordered forms are presented in Figure 7.5.

Figure 7.5: Reordered forms of neos5 obtained by applying GCG decomposition, block ordering, and recursive rules

To evaluate the impact of reordering, we compute two sets of metrics: permutation distances and the standard deviation of work units.

First, we consider the pairwise permutation distances between the coefficient matrices. For each pair of matrices (A_i, A_j), where $i \neq j$, we calculate a distance $d(A_i, A_j)$ using the Kendall Tau distance metric. Given four matrices, this yields $\binom{4}{2} = 6$ distance values. We then compute the standard deviation of these distances:

$$\sigma_{\text{dist,perm}} = \sqrt{\frac{1}{6}\sum_{i<j}\left(d(A_i, A_j) - \mu_{\text{perm}}\right)^2} \quad \text{and} \quad \sigma_{\text{dist,reord}} = \sqrt{\frac{1}{6}\sum_{i<j}\left(d(A'_i, A'_j) - \mu_{\text{reord}}\right)^2},$$

where μ_{perm} and μ_{reord} are the respective means of the distances for the original and reordered matrices. The results are $\sigma_{\text{dist,perm}} = 314$ and $\sigma_{\text{dist,reord}} = 310$.

Secondly, we analyze the standard deviation of work units across different permutations, based on the recorded solve times. Let $t_1, \ldots, t_4$ be the solve times for the original permutations and $t'_1, \ldots, t'_4$ for the reordered versions. We compute:

$$\sigma_{\text{time,perm}} = \sqrt{\frac{1}{4}\sum_{k=1}^{4}(t_k - \bar{t}_{\text{perm}})^2} \quad \text{and} \quad \sigma_{\text{time,reord}} = \sqrt{\frac{1}{4}\sum_{k=1}^{4}(t'_k - \bar{t}_{\text{reord}})^2},$$

where $\bar{t}_{\text{perm}}$ and $\bar{t}_{\text{reord}}$ are the respective mean solve times. The standard deviation before reordering is $\sigma_{\text{time,perm}} = 19.76$, and after reordering it is reduced to $\sigma_{\text{time,reord}} = 8.7$.

In this case, we observe a significant reduction in solving time variability, as indicated by the drop in standard deviation from $\sigma_{\text{time,perm}} = 19.76$ to $\sigma_{\text{time,reord}} = 8.7$, demonstrating the effectiveness of the reordering in stabilizing solver effort across permutations.

8 | Pseudocode Reference for Rules Implementation

In this chapter, we present a collection of heuristic rules designed to score and reorder constraints and variables in mixed-integer programming (MIP) problems. These rules are based on structural features such as variable types, finite bounds, sparsity patterns, and objective coefficients. By computing rule-specific scores, we influence the ordering of rows and columns in the constraint matrix to uncover meaningful substructures, reduce permutation variance, and ultimately improve solver robustness or preprocessing efficiency.

Earlier, we explained how our ordering algorithms are composed and applied. Here, we provide a detailed explanation of the individual rules referenced previously. Each rule may impact both constraint and variable scores, which is why we report, for each rule, two high-level pseudocode functions, one for scoring columns (variables) and one for scoring rows (constraints). Each rule targets a specific characteristic of the problem instance and can be applied independently or in combination with others.

8.1. Shared Rules Pseudocode

The following rules are shared by both the hierarchical and recursive approaches to score variables and constraints based on structural features of the problem.

8.1.1. VariableTypeRule

This rule assigns scores to variables based on their types. It reflects the modeling complexity introduced by different variable types.

Algorithm 8.1: score variables

Require: The number of variables, their types, and a 2D array of lower and upper bounds

1: Compute the difference between the upper and lower bounds
2: Initialize the score of each variable to zero.

3: **For each variable:**
   1. **If** the variable is SEMICONT, assign a score of 5.
   2. **Else If** the variable is SEMIINT, assign a score of 4.
   3. **Else if** the variable is BINARY, assign a score of 3.
   4. **Else if** the variable is CONTINUOUS, assign a score of 1.
   5. **Else if** the variable is INTEGER:
      - **If** the bound difference is approximately 1, assign a score of 3.
      - **Else** assign a score of 2.
   6. **Else** assign a score of 0.
4: Multiply all variable scores by the scaling factor.
5: **return** the vector of variable scores.

This rule did not assign any score to constraints, so a zero vector was returned for them.

8.1.2. BoundCategoryRule

This rule scores variables according to the nature of their bounds. Variables with finite or one-sided bounds receive higher scores, as their domains more strongly influence the feasible region.

Algorithm 8.2: score variables method

Require: A 2D array of finite lower and upper bounds $(\text{lb}_i, \text{ub}_i)$ for each variable

1: **if** the variable has finite bounds **then**
2: **if** lb ≥ 0 or ub ≤ 0, assign a score of 4.
3: **else if** lb < 0 and ub > 0, assign a score of 3.
4: **else** assign a fallback score of 2.
5: **end if**
6: For variables with exactly one infinite bound, assign a score of 2.
7: For variables with both bounds infinite, assign a score of 1.
8: Multiply the computed scores by the scaling factor.
9: **return** the vector of variable scores.

This rule did not assign any score to constraints, so a zero vector was returned for them.

8.1.3. ConstraintCompositionRule

This rule scores constraints based on the types of variables they involve. Constraints with only integer or only continuous variables are considered more structurally uniform and receive higher scores than mixed-type constraints.
This rule did not assign any score to variables, so a zero vector was returned for them.

Algorithm 8.3: score constraints method

Define a mapping for variable types:
- Integral types (BINARY, INTEGER, SEMIINT) → 3,
- Continuous types (CONTINUOUS, SEMICONT) → 2.

For each variable, assign its corresponding code based on its type.
for each constraint (each row in A using the CSR representation) **do**
    Retrieve the indices of nonzero entries in the current row.
    if the row has no nonzero entries **then**
        Set the constraint's score to $1 \times$ scaling.
    else
        Extract the type codes for the variables present in the row.
        if all extracted codes equal 3 **then**
            Set the constraint's score to $3 \times$ scaling.
        else if all extracted codes equal 2 **then**
            Set the constraint's score to $2 \times$ scaling.
        else
            Set the constraint's score to $1 \times$ scaling.
        end if
    end if
end for
return the vector of constraint scores.

8.2. Hierarchical Rules Pseudocode

The following rules are used exclusively in the hierarchical approach to score variables and constraints based on structural features of the problem.

8.2.1. ColumnsCoefficientRule

The score for each variable was computed as:

$$\text{score}_j = \alpha \cdot \sum_i \log\left(1 + |a_{ij}|\right)$$

where α is the scaling factor and a_{ij} is the coefficient of variable j in constraint i.
This rule did not assign any score to constraints, so a zero vector was returned for them.

8.2.2. ObjectiveCoefficientRule

The score for each variable was computed as:

$$\text{score}_j = \alpha \cdot \log\left(1 + |c_j|\right)$$

where α is the scaling factor and c_j is the objective coefficient of variable j.
This rule did not assign any score to constraints, so a zero vector was returned for them.

8.2.3. VariableOccurrenceRule

This rule rewards variables that appear in more constraints, indicating their structural importance within the model.

Algorithm 8.4: score variables method

1: Check if the coefficient matrix A is sparse.
2: **if** A is sparse **then**
3: Convert A to a boolean matrix.
4: Sum the boolean values in each column to count nonzero entries.
5: **else**
6: Directly count the nonzero entries in each column.
7: **end if**
8: Multiply the resulting column counts by the scaling factor.
9: **return** the vector of scores.

This rule did not assign any score to constraints, so a zero vector was returned for them.

8.2.4. ConstraintSenseRule

This rule ranks constraints by their sense, assigning different priorities to equality, less-than, and greater-than constraints to reflect their structural roles.
This rule did not assign any score to variables, so a zero vector was returned for them.

Algorithm 8.5: score constraints method

1: Define a mapping from constraint senses to priorities: {“$\leq$” : 1, “$=$” : 2, “$\geq$” : 3}.
2: For each constraint, retrieve its sense.
3: Look up the corresponding numeric priority (defaulting to 0 if the sense is not found).
4: Multiply the priority by the scaling factor.
5: **return** the vector of computed constraint scores.

8.2.5. RowCoefficientRule

This rule did not assign any score to variables, so a zero vector was returned for them. The score for each constraint i was computed as:

$$\text{score}_i = \alpha \cdot \sum_j \log\left(1 + |a_{ij}|\right)$$

where α is the scaling factor and a_{ij} is the coefficient of variable j in constraint i.

8.2.6. RHSValueRule

This rule did not assign any score to variables, so a zero vector was returned for them. The score for each constraint i was computed as:

$$\text{score}_i = \alpha \cdot \log\left(1 + |b_i|\right)$$

where α is the scaling factor and b_i is the right-hand side value of constraint i.

8.2.7. ConstraintRangeRule

This rule evaluates constraints based on the range of their coefficients, capturing the spread of influence exerted by variables in each row.
This rule did not assign any score to variables, so a zero vector was returned for them.

Algorithm 8.6: score constraints method

1: Initialize an empty list `ranges`.
2: **for** each row in the coefficient matrix A **do**
3: Compute the maximum value in the row.
4: Compute the minimum value in the row.
5: Calculate the range as: range = max − min.
6: Append range to `ranges`.
7: **end for**
8: For each value in `ranges`, compute $\log(1 + |\text{range}|)$.
9: Multiply each result by the scaling factor.
10: **return** the vector of computed scores.

8.3. Recursive Rules Pseudocode

The following rules are specific to the recursive approach and are designed to guide block decomposition based on substructure characteristics.

8.3.1. AllBinaryVariablesRule

This rule scores rows and columns based on whether they exclusively involve binary variables.

Algorithm 8.7: score variables method

Convert the list of bounds into a 2D array of shape (n_vars, 2).
For each variable, check if its lower bound is approximately 0 and its upper bound is approximately 1 (using the given tolerance).
Mark the variable as binary if both conditions are met.
Assign a score of `scaling` to binary variables; otherwise, assign a score of 0.
return the vector of computed variable scores.

Algorithm 8.8: score constraints method

Determine the number of constraints from the coefficient matrix A.
Convert the list of bounds into a 2D array.
Create a boolean array for all variables, marking each as binary if its lower bound is close to 0 and its upper bound is close to 1.
Initialize a zero vector of length equal to the number of constraints.
if A is in sparse (CSR) format **then**
    for each constraint i **do**
        Retrieve the column indices of nonzero entries in row i using A_csr.
        if the row has at least one nonzero entry and all corresponding variables are binary **then**
            Set the score for constraint i to `scaling`.
        end if
    end for
else
    for each constraint i **do**
        Extract the ith row of A as a dense vector.
        Identify the indices of nonzero entries using the tolerance.
        if the row is nonempty and all corresponding variables are binary **then**
            Set the score for constraint i to `scaling`.

18: **end if**
19: **end for**
20: **end if**
21: **return** the vector of computed constraint scores.

8.3.2. AllCoefficientsOneRule

This rule scores rows and columns where all nonzero coefficients are approximately one.

Algorithm 8.9: score variables method

1: Let `num_vars` be the number of columns in A.
2: **if** A supports CSC format **then**
3: Obtain the column pointer array from `A_csc.indptr`.
4: **for** each column j (from 0 to `num_vars` - 1) **do**
5: Compute the number of nonzero entries: `count` = `indptr`$[j+1]$ − `indptr`$[j]$.
6: Extract the nonzero data for column j from `A_csc.data`.
7: Compute the absolute deviation $|a-1|$ for each nonzero entry.
8: Let `max_diff`$[j]$ be the maximum deviation in column j;
if the column is empty, set `max_diff`$[j] = \infty$.
9: **If** `max_diff`$[j] \leq$ `tol`, then set the score for column j to `scaling`; otherwise, set it to 0.
10: **end for**
11: **else**
12: **for** each column j in the dense matrix A **do**
13: Extract column j.
14: Identify nonzero entries where $|a| >$ `tol`.
15: **If** nonzero entries exist and all are approximately 1 (within `tol`), then set score for column j to `scaling`; else, set it to 0.
16: **end for**
17: **end if**
18: **return** the vector of variable scores.

Algorithm 8.10: score constraints method

1: Let `num_constraints` be the number of rows in A.
2: **if** A supports CSR format **then**
3: Obtain the row pointer array from `A_csr.indptr`.
4: **for** each row i (from 0 to `num_constraints` - 1) **do**
5: Compute the number of nonzero entries: `count` = `indptr`$[i+1]$ − `indptr`$[i]$.

Extract the nonzero data for row i from `A_csr.data`.
Compute the absolute deviation $|a - 1|$ for these entries.
Let `max_diff`$[i]$ be the maximum deviation in row i;
if row i is empty, set `max_diff`$[i] = \infty$.
If `max_diff`$[i] \leq$ `tol`, then set the score for row i to `scaling`; otherwise, set it to 0.
end for
else
for each row i in the dense matrix A **do**
Extract row i.
Identify nonzero entries where $|a| >$ `tol`.
If nonzero entries exist and all are approximately 1 (within `tol`), then set score for row i to `scaling`; else, set it to 0.
end for
end if
return the vector of constraint scores.

8.3.3. NonZeroCountRule

This rule assigns scores based on the number of nonzero entries in each row or column.

Algorithm 8.11: score variables and score constraints methods

Determine the dimensions of the coefficient matrix.
if the matrix is stored in sparse column (CSC) format **then**
Use the CSC index pointers to locate the stored entries.
Create a mask to flag entries with absolute value greater than the tolerance.
Count the flagged entries for each column.
else
Create a mask for the entire dense matrix to flag entries with absolute value above the tolerance.
Sum the mask along each column to count nonzero entries.
end if
Multiply the resulting counts by the scaling factor.
return the score vector.

8.3.4. SignPatternRule

This rule scores constraints based on the pattern of positive and negative coefficients, adjusted for constraint type.
This rule did not assign any score to variables, so a zero vector was returned for them.

Algorithm 8.12: score constraints method

1: Determine the dimensions of the coefficient matrix A.
2: **if** A can be converted to a dense array **then**
3: Convert A to a dense array A_{dense}.
4: **else**
5: Use A as is.
6: **end if**
7: For each row in A_{dense}:
   1. Count the number of entries greater than the tolerance (tol); store as the positive count.
   2. Count the number of entries less than $-tol$; store as the negative count.
8: For each corresponding constraint, examine its string representation:
   - **If** the constraint contains "$\leq$", set its slack sign to "+".
   - **If** the constraint contains "$\geq$", set its slack sign to "$-$".
   - **Else**, no adjustment is made.
9: Add 1 to the positive count for rows with slack sign "+".
10: Add 1 to the negative count for rows with slack sign "$-$".
11: For each row, set the final score as the maximum of the adjusted positive and negative counts.
12: **return** the vector of final scores.

8.3.5. ConstraintIntegerCountRule

This rule did not assign any score to variables, so a zero vector was returned for them.

Algorithm 8.13: score constraints method

1: Determine the dimensions (number of constraints and variables) from the coefficient matrix A.
2: For each variable, assign an indicator:
   - If the variable's type is BINARY, INTEGER, or SEMIINT, set indicator = 1.
   - Otherwise, set indicator = 0.
3: **if** the matrix A is in CSR (sparse) format **then**

Create a copy of A in CSR format and replace all nonzero entries with 1 to form a binary matrix.
Multiply each column of the binary matrix by the corresponding variable indicator.
For each row, count the nonzero entries; assign this count as the constraint's score.
else
Convert A to a dense matrix.
Create a binary matrix by setting each element to 1 if its absolute value exceeds a given tolerance, and 0 otherwise.
Multiply each column by its corresponding variable indicator.
Sum the entries in each row; assign this sum as the constraint's score.
end if
return the vector of constraint scores.

8.3.6. ConstraintContinuousCountRule

This rule did not assign any score to variables, so a zero vector was returned for them.

Algorithm 8.14: score constraints method

Precompute an indicator vector for continuous variables:
for each variable **do**
    if variable type is CONTINUOUS or SEMICONT **then**
        Set indicator to 1.
    else
        Set indicator to 0.
    end if
end for
Set tolerance tol (e.g., 1×10^{-15}).
if A is in sparse CSR format **then**
    Create a binary version of A by setting each nonzero entry to:
    - 1, if $|\text{entry}| > tol$;
    - 0, otherwise.
    Compute the score for each row as the dot product of the binary matrix with the indicator vector.
else
    Convert A to a dense matrix.
    Create a binary mask where each element is 1 if $|\text{entry}| > tol$, and 0 otherwise.
    Multiply the binary mask element-wise by the indicator vector.
    Sum the results along each row to obtain the score.

18: **end if**
19: **return** the vector of computed scores for the constraints.

8.3.7. BothBoundsFiniteCountRule

Algorithm 8.15: score variables method

1: Convert the list of bounds into a two-dimensional array of shape (n_vars, 2).
2: For each variable, check if both the lower and upper bounds are finite.
3: **If** both bounds are finite, assign a score of 1; otherwise, assign a score of 0.
4: Multiply the resulting score vector by the scaling factor.
5: **return** the vector of variable scores.

Algorithm 8.16: score constraints method

1: Convert the list of bounds into a two-dimensional array.
2: Create an indicator vector for variables: 1 if both bounds are finite, 0 otherwise.
3: **if** the coefficient matrix A is in sparse (CSR) format **then**
4: Copy A and convert its nonzero entries to 1, forming a binary matrix.
5: Multiply each column of the binary matrix by the corresponding indicator value.
6: For each constraint (row), sum the resulting values to compute its score.
7: **else**
8: Convert A to a dense matrix.
9: Create a binary mask where each element is 1 if it is nonzero, and 0 otherwise.
10: Multiply each column by the indicator vector.
11: Sum the mask values along each row to obtain the score.
12: **end if**
13: **return** the vector of constraint scores.

8.3.8. BothBoundsInfiniteCountRule

Algorithm 8.17: score variables method

1: Convert the list of bounds into a 2D array of shape (n_vars, 2).
2: For each variable, check whether both the lower and upper bounds are infinite.
3: Create a boolean mask where each element is True if both bounds are infinite, False otherwise.
4: Convert the boolean mask to integers (1 for True, 0 for False).
5: Multiply the resulting vector by the scaling factor.
6: **return** the vector of variable scores.

Algorithm 8.18: score constraints method

Convert the list of bounds into a 2D array.
Create an indicator vector for variables: set to 1 if both bounds are infinite, 0 otherwise.
if the coefficient matrix A is in CSR (sparse) format **then**
    Make a copy of A in CSR format.
    Convert its nonzero entries to 1, forming a binary matrix.
    For each nonzero entry, multiply it by the corresponding variable indicator using its column index.
    Sum the modified entries along each row to compute a preliminary score for each constraint.
else
    Convert A to a dense matrix.
    Create a binary mask: each element is 1 if its absolute value is nonzero, 0 otherwise.
    Multiply each column of the binary matrix by the corresponding indicator value.
    Sum the results along each row to compute the score for each constraint.
end if
Multiply the computed scores by the scaling factor.
return the vector of constraint scores.

8.3.9. OneBoundFiniteCountRule

Algorithm 8.19: score variables method

Convert the list of bounds into a 2D array of shape (n_vars, 2).
For each variable, create a boolean mask indicating finite bounds (i.e., not infinite).
Sum the mask values for each variable to count the number of finite bounds (resulting in 0, 1, or 2).
For each variable, assign a score of 1 if exactly one bound is finite; otherwise, assign a score of 0.
Multiply the score vector by the scaling factor.
return the vector of variable scores.

Algorithm 8.20: score constraints method

Convert the list of bounds into a 2D array.
Create an indicator vector for variables by:
- Checking for each variable if exactly one bound is finite (assign 1 if true, 0 otherwise).

3: **if** the coefficient matrix A is in sparse (CSR) format **then**
4: Copy A into a CSR matrix.
5: Convert its nonzero entries to 1, forming a binary matrix.
6: Multiply each nonzero entry by the corresponding variable's indicator (using the column indices).
7: Sum the modified entries along each row to compute each constraint's score.
8: **else**
9: Convert A to a dense matrix.
10: Create a binary mask where each element is 1 if it is nonzero, and 0 otherwise.
11: Multiply each column of the binary mask by the corresponding variable's indicator.
12: Sum the results along each row to compute each constraint's score.
13: **end if**
14: Multiply the resulting constraint scores by the scaling factor.
15: **return** the vector of constraint scores.

8.4. Binary Rules Pseudocode

The following special rules are applied only when the problem contains binary variables exclusively, leveraging properties unique to purely binary formulations.

8.4.1. SetPackingRHSRule

This rule scores constraints involving only binary variables by considering their right-hand side values, and scores binary variables with a fixed weight.

Algorithm 8.21: score variables method

1: Convert the list of bounds into a 2D array of shape (n_vars, 2).
2: For each variable, check if the lower bound is approximately 0 and the upper bound is approximately 1 (using the specified tolerance).
3: **If** both conditions hold for a variable, mark it as binary.
4: For each variable, assign a score of `scaling` if it is binary; otherwise, assign a score of 0.
5: **return** the vector of variable scores.

Algorithm 8.22: score constraints method

1: Let n be the number of constraints (rows in A).
2: Convert the list of bounds into a 2D array.
3: For each variable, determine its binary status by checking if its lower bound is close

to 0 and its upper bound is close to 1.
Initialize a vector of zeros for the constraint scores.
if A is in sparse (CSR) format **then**
    for each constraint $i = 1, \ldots, n$ **do**
        Retrieve the column indices of nonzero entries in row i.
        If the constraint involves at least one variable and all corresponding variables are binary,
        Set the score for constraint i to `scaling` $\times$ rhs$[i]$.
    end for
else
    for each constraint $i = 1, \ldots, n$ **do**
        Extract the ith row of A and identify indices of nonzero coefficients (using the tolerance).
        If the constraint involves at least one variable and all involved variables are binary,
        Set the score for constraint i to `scaling` $\times$ rhs$[i]$.
    end for
end if
return the vector of constraint scores.

8.4.2. UnscaledObjectiveOrderingRule

This rule scores variables proportionally to their objective coefficients if the constraint matrix is approximately unscaled (i.e., all nonzero entries are close to one).

Algorithm 8.23: score variables method

Check if instance is unscaled:
if the coefficient matrix A is in sparse (CSR) format **then**
    If A has no nonzero entries, consider it unscaled.
    Else check that every nonzero entry in A is approximately 1 (within tolerance).
else
    For a dense matrix, verify that all nonzero entries are approximately 1 (within tolerance).
end if
If the matrix is unscaled:
- Compute each variable's score as: `scaling` $\times$ objective coefficient.

Else:
- Set the score for every variable to 0.

10: **return** the vector of computed variable scores.

This rule did not assign any score to constraints, so a zero vector was returned for them.

Conclusions

After exploring various approaches to reduce variability and stabilize solver performances, we can summarize our results and lay the foundation for future research that will pursue the objective to improve and have a better understanding of what we described in the previous chapters. First of all our aim was to mitigate a possible adversarial behaviour consisting of permuting rows and columns of a problem that perturbs the existing structure of its formulation.

The experiments outcome suggests that in the case of a permutation of all rows and columns, which creates a completely random structure inside the matrix, we did not find any new method able to create a reordering effectively. Also, sophisticated structure detection frameworks, such as the one built into GCG, have similar results.

On the other hand, when we consider permutations built with coarser granularities, both our algorithm based on rules and the structure detection performed by GCG have a significant impact in reducing variability. Based on these results, we can conclude that it could be worth further refining the algorithms we described in the previous chapter to obtain a production version of them and work on an integration into a commercial solver, as it is very likely that most of the problems formulated starting from real-world scenarios do not have a completely random structure but more likely structures that are comparable to the coarser permutations that we found are positively impacted by our algorithm. One point that would be interesting to explore more, and could be useful to get a better understanding of which structures are causing variability and which ones are responsible for more stable results, is to define a distance metric that truly correlates with variability.

From the experiments results we eventually found out that even if intuitively and from the initial information we had our metric based on Kendall Tau distance could have been a great choice, we did not get the expected correlation in the experiment results.

In the end, we should remember that very few studies have been led so far in this specific field, consequently, there is room for further investigations on better techniques to reduce performance variability that may face the problem following completely different paths.

Bibliography

[1] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. Finding cuts in the TSP (a preliminary report). In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 20, pages 1–20. American Mathematical Society, Providence, RI, 1995.

[2] C. Aykanat, A. Pınar, and V. Çatalyürek. Permuting Sparse Rectangular Matrices into Block-Diagonal Form. *SIAM Journal on Scientific Computing*, 25(6):1860–1879, Jan. 2004. ISSN 1064-8275, 1095-7197. doi: 10.1137/S1064827502401953.

[3] M. Bergner, A. Caprara, A. Ceselli, F. Furini, M. E. Lübbecke, E. Malaguti, and E. Traversi. Automatic Dantzig–Wolfe reformulation of mixed integer programs. *Mathematical Programming*, 149(1-2):391–424, Feb. 2015. ISSN 0025-5610, 1436-4646. doi: 10.1007/s10107-014-0761-5.

[4] P. Bonami, D. Salvagnin, and A. Tramontani. Implementing Automatic Benders Decomposition in a Modern MIP Solver. In D. Bienstock and G. Zambelli, editors, *Integer Programming and Combinatorial Optimization*, volume 12125, pages 78–90. Springer International Publishing, Cham, 2020. ISBN 978-3-030-45770-9 978-3-030-45771-6. doi: 10.1007/978-3-030-45771-6_7. Series Title: Lecture Notes in Computer Science.

[5] F. Clautiaux and I. Ljubić. Last fifty years of integer linear programming: A focus on recent practical advances. *European Journal of Operational Research*, Nov. 2024. ISSN 0377-2217. doi: 10.1016/j.ejor.2024.11.018.

[6] FICO. Fico® Xpress Optimization, 2025. URL `https://www.fico.com/en/products/fico-xpress-optimization`. Accessed: 2025-03-20.

[7] C.-O. Foundation. Decomposition for integer programming, 2025. URL `https://github.com/coin-or/Dip`. Accessed: 2025-04-04.

[8] GCG. Generic column generation: Main page, 2025. URL `https://gcg.or.rwth-aachen.de/doc-3.5.0/index.html`. Accessed: 2025-03-20.

[9] Gurobi Optimization. Gurobipy, 2025. URL https://pypi.org/project/gurobipy/. Accessed: 2025-06-04.

[10] Gurobi Optimization. Gurobi optimizer, 2025. URL https://www.gurobi.com/solutions/gurobi-optimizer. Accessed: 2025-03-20.

[11] Gurobi Optimization. Model attributes - gurobi optimizer reference manual, 2025. URL https://docs.gurobi.com/projects/optimizer/en/current/reference/attributes/model.html. Accessed: 2025-03-20.

[12] IBM. Ibm ilog cplex optimization studio, 2025. URL https://www.ibm.com/products/ilog-cplex-optimization-studio. Accessed: 2025-04-04.

[13] M. G. Kendall. A New Measure of Rank Correlation. *Biometrika*, 30(1/2):81–93, 1938. ISSN 00063444. doi: 10.2307/2332226. Publisher: Oxford University Press, Biometrika Trust.

[14] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter. MIPLIB 2010: Mixed Integer Programming Library version 5. *Mathematical Programming Computation*, 3(2):103–163, June 2011. ISSN 1867-2949, 1867-2957. doi: 10.1007/s12532-011-0025-9.

[15] A. Lodi and A. Tramontani. Performance Variability in Mixed-Integer Programming. In *Theory Driven by Influential Applications*, INFORMS TutORials in Operations Research, pages 1–12. INFORMS, Sept. 2013. ISBN 978-0-9843378-4-2. doi: 10.1287/educ.2013.0112. Section: 1.

[16] S. Manikandan. Measures of dispersion. *Journal of Pharmacology and Pharmacotherapeutics*, 2(4):315–316, Dec. 2011. ISSN 0976-500X, 0976-5018. doi: 10.4103/0976-500X.85931.

[17] Matplotlib Development Team. Matplotlib, 2025. URL https://matplotlib.org/. Accessed: 2025-06-04.

[18] MIPLIB. The benchmark set, 2017. URL https://miplib.zib.de/tag_benchmark.html. Accessed: 2025-04-04.

[19] MIPLIB. Miplib - mixed integer problem library, 2017. URL https://miplib.zib.de. Accessed: 2025-04-04.

[20] NumPy Developers. Numpy, 2025. URL https://numpy.org/. Accessed: 2025-06-04.

[21] SCIP. Scip doxygen documentation: How to provide a problem decomposition, 2025. URL `https://www.scipopt.org/doc/html/DECOMP.php`. Accessed: 2025-04-04.

[22] SciPy Community. Scipy, 2025. URL `https://scipy.org/`. Accessed: 2025-06-04.

[23] Streamlit Inc. Streamlit, 2025. URL `https://streamlit.io/`. Accessed: 2025-06-04.

[24] The pandas Development Team. pandas, 2025. URL `https://pandas.pydata.org/`. Accessed: 2025-06-04.

[25] M. Waskom. Seaborn, 2025. URL `https://seaborn.pydata.org/`. Accessed: 2025-06-04.

List of Figures

List of Tables

Ringraziamenti

Desidero esprimere la mia gratitudine al Prof. Pietro Belotti, relatore di questa tesi, per la disponibilità e i consigli che hanno orientato e arricchito il mio lavoro. Un grazie va alla mia famiglia, per il sostegno costante e l'incoraggiamento che non sono mai mancati lungo il percorso. Un pensiero va anche ai miei amici, che con la loro presenza hanno saputo alleggerire i momenti difficili e ricordarmi l'importanza dell'equilibrio tra impegno e leggerezza.