# EFFICIENT HEURISTICS FOR THE CONSECUTIVE ZEROS PROBLEM IN SPARSE JACOBIAN MATRIX DETERMINATION

**MD. ASIF TALUKDAR**
**Bachelor of Science, BRAC University, 2020**

A thesis submitted
in partial fulfillment of the requirements for the degree of

**MASTER OF SCIENCE**

in

**COMPUTER SCIENCE**

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

© Md. Asif Talukdar, 2025

EFFICIENT HEURISTICS FOR THE CONSECUTIVE ZEROS PROBLEM IN SPARSE JACOBIAN MATRIX DETERMINATION

MD. ASIF TALUKDAR

Date of Defence: December 17, 2025

| | | |
|---|---|---|
| Dr. Shahadat Hossain<br>Thesis Co-Supervisor | Professor | Dr. Scient |
| Dr. Robert Benkoczi<br>Thesis Co-Supervisor | Professor | Ph.D. |
| Dr. Saurya Das<br>Thesis Examination Committee Member | Professor | Ph.D. |
| Dr. Pascal Ghazalian<br>Thesis Examination Committee Member | Professor | Ph.D. |
| Dr. Wendy Osborn<br>Chair, Thesis Examination Committee | Associate Professor | Ph.D. |

# Dedication

This thesis is dedicated to my beloved parents. No acknowledgment is enough for their unwavering support, sacrifice, and for believing in me more than myself; some connections are above anything, and some debts are beyond repayment.

# Abstract

We study the Consecutive Zeros Problem (CZP), which is an extension of the well-known maximum consecutive ones problem. CZP is interesting both from a combinatorial and a practical standpoint. Examples of practical applications include sparse Jacobian matrix determination [12, 13] and cache-friendly sparse linear algebra [8]. Leveraging a priori known sparsity pattern of matrix $A \in \mathbb{R}^{m \times n}$, we employ a state-of-the-art coloring/column partitioning algorithm to generate a seed matrix $S \in \{0,1\}^{n \times C_n}$, to obtain the compressed matrix $B \in \mathbb{R}^{m \times C_n} = A \times S$, using automatic differentiation (AD) or finite differences (FD) while significantly reducing the problem's dimensionality. With AD forward mode, nonzero entries of $A$ are mapped to the compressed matrix accurately up to the machine precision using $C_n$ "forward passes". This research aims at reducing the computation cost ( measured by the number of AD passes required to compress the matrix) further by solving the CZP (grouping the zeros in each row of $B$). The problem is cast as a variant of the Traveling Salesperson Problem (TSP). To solve the problem, we have designed and implemented heuristic algorithms, including Ant Colony Optimization (ACO) meta-heuristic and two variants of ACO (Zp-Hybrid and Zp-Estimator), and several greedy heuristics (Exhaustive Approach, Selective Exhaustive, and Single Attempt). The computational results from a subset of benchmark test matrices clearly demonstrate the effectiveness of the proposed approaches.

# Acknowledgments

I want to thank almighty Allah for blessing me with the opportunity to complete my research-based Master's. I also want to thank Allah for granting me a great supervisor, Dr. Shahadat Hossain. My thesis is unimaginable without him; the amount of support he provided academically and also for personal well-being is unparalleled. I believe it would be difficult for me to work with other researchers after this. My sincere gratitude to my co-supervisor, Dr. Robert Benkoczi, for all the advice and guidance; his mentorship can be an example. I want to thank Dr. Wendy Osborn for all the support during my Master's, and I truly appreciate all the help. I am thankful to my thesis committee members, Dr. Saurya Das and Dr. Pascal Ghazalian, for their guidance and suggestions. My support system in Lethbridge, Hasib, Emam, Maowa, and Mahmud, I thank you all for being by my side during this journey. I am grateful to my dear friends Ayesha and Noshin for their encouragement to undertake this path and for their motivation during my Master's studies. Lastly, I would like to express my gratitude to the School of Graduate Studies and Alberta Innovates for their financial and administrative support throughout my Master's program.

# Contents

**Dedication** iii

**Abstract** iv

**Acknowledgments** v

**List of Tables** viii

**List of Figures** ix

**List of Algorithms** x

**1 Introduction** 1
- 1.1 Related Work . . . . . . . . 2
- 1.2 Our contribution . . . . . . . . 4
- 1.3 Thesis Organization . . . . . . . . 5

**2 Background and Methodology** 7
- 2.1 Sparse Matrix . . . . . . . . 7
- 2.2 Jacobian Matrix . . . . . . . . 8
- 2.3 Newton's Method . . . . . . . . 9
- 2.4 Direct Determination . . . . . . . . 11
- 2.5 Compressed Sparse Row (CSR) Format . . . . . . . . 11
- 2.6 Matrix Partitioning and Seed Matrix Computation . . . . . . . . 13
- 2.7 Substitution methods . . . . . . . . 16
  - 2.7.1 Basic two–group substitution . . . . . . . . 18
  - 2.7.2 Generalized $Z_p$–group substitution . . . . . . . . 20
  - 2.7.3 Illustrative example . . . . . . . . 20
- 2.8 Consecutive Zeros Problem . . . . . . . . 21
- 2.9 Implementation Process . . . . . . . . 22
  - 2.9.1 Illustrative Example . . . . . . . . 24

**3 Greedy Heuristics** 27
- 3.1 Improve Zp permutation . . . . . . . . 27
- 3.2 Illustrative Example . . . . . . . . 31
- 3.3 Exhaustive Approach . . . . . . . . 32
- 3.4 Selective Exhaustive Approach . . . . . . . . 33
- 3.5 Single Attempt Approach . . . . . . . . 33

**4 Ant Colony Optimization** **38**
4.1 TSP formulation & cost distance . . . 38
4.2 Cost Array Calculation . . . 39
4.3 Matrix Input . . . 40
4.4 ACO Algorithm Overview . . . 41
4.4.1 Algorithm for input parsing . . . 42
4.4.2 Algorithm for Distance Array . . . 42
4.4.3 Nearest-neighbor candidate lists . . . 44
4.4.4 Pheromone and distance heuristic initialization . . . 46
4.4.5 Solution construction (roulette on $S_i$) . . . 47
4.4.6 Local search . . . 48
4.4.7 MMAS pheromone model . . . 49
4.4.8 Termination Criteria . . . 50
4.4.9 Complexity and memory . . . 50

**5 Hybrid ACO and Zp Estimator** **51**
5.1 Zp-Hybrid . . . 51
5.1.1 Calculating Zpair . . . 51
5.1.2 Zpair-based nearest-neighbor lists . . . 53
5.1.3 Hybrid Neighbor Lists . . . 54
5.1.4 Zp-aware Solution Construction . . . 55
5.1.5 Zp-aware Pheromone Update . . . 57
5.2 Zp-Estimator . . . 59
5.2.1 First city initialization and parameter update . . . 59
5.2.2 Zp-Estimator Solution Construction . . . 60
5.2.3 Zp Finalize tour . . . 63

**6 Experimental Evaluation** **65**
6.1 Greedy Heuristic Output . . . 66
6.2 ACO Superiority on Constructed Matrices . . . 66
6.3 Update Over Nearest Neighbor Heuristic . . . 68
6.4 Overall Comparison of Heuristics . . . 69

**7 Conclusion and Future Works** **71**
7.1 Future Research Directions . . . 71

**Bibliography** **73**

# List of Tables


4.1 ACO variants at a glance (we focus on MMAS) . . . . . . . . . . . . . . . . 41

5.1 Zp–specific fields stored per ant (`ant_struct`) . . . . . . . . . . . . . . . . . . 59

6.1 Default parameters used in our ACO implementation . . . . . . . . . . . . . 67
6.2 Test Results on `Constructed Matrices`, employing `ACO` ($d_3$), `Hybrid-Zp` ($d_7$) and `Zp-Estimator` ($d_8$) . . . . . . . . . . . . . . . . . . . . . . . . . . 68
6.3 Comparison of Nearest Neighbor Heuristic and 2-opt algorithm with our heuristics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
6.4 Overall performance of the heuristics . . . . . . . . . . . . . . . . . . . . . . 70

# List of Figures

2.1 Structure plot of sparse matrix bcsstm26 (Dimensions: 1922 × 1922, 1922 non-zero entires). Source [4] . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.2 CSR arrays: `val`, `colIdx`, `rowPtr`. . . . . . . . . . . . . . . . . . . . . . . . . 13

# List of Algorithms


2.1 Newton's method . . . . . 10
2.2 CSR Matrix Multiplication . . . . . 16
2.3 Efficient Reconstruction of Jacobian Matrix . . . . . 23
3.1 Find-Longest-Consecutive-Zeros . . . . . 28
3.2 Evaluate-Permutation . . . . . 29
3.3 Compute-Zt-Zi-Permuted . . . . . 30
3.4 Get-Prioritized-Candidates . . . . . 31
3.5 Improve-Zp-Permutation . . . . . 34
3.6 Try-Improve-Row-Exhaustive . . . . . 35
3.7 Try-Improve-Row-Selective-Exhaustive . . . . . 36
3.8 Try-Improve-Row-Single-Attempt . . . . . 37
4.1 ACO Main: Overall workflow . . . . . 42
4.2 ReadMatrix: Parse custom .mtx input . . . . . 43
4.3 ComputeDistances: Symmetric inter-city distance array . . . . . 44
4.4 MatrixDistance(i, j): Row-wise difference count between columns . . . . . 45
4.5 Build Nearest Neighbor Lists . . . . . 45
4.6 Construct-Solutions: Build tour for ants . . . . . 48
5.1 ComputeZpair: Upper-triangular co-zero counts . . . . . 53
5.2 BuildNNLists Zpair(k): Top-k neighbors per column by ZPair . . . . . 54
5.3 BuildNNLists Hybrid(kdist, kzp): Merge Zpair-NN and distance lists . . . . . 56
5.4 ComputeZpForTour: Wrap-aware $Z_p$ for a finished permutation . . . . . 57
5.5 Global Update Pheromone Add: Reinforce pheromone based on tour . . . . . 58
5.6 Dzp init after first city: initialize per–row Zp state . . . . . 60
5.7 Dzp_projected_Zp_if_append: look-ahead $Z_p$ if appending *candidate* . . . . . 61
5.8 Dzp_update_after_append . . . . . 62
5.9 Dzp_finalize_tour: compute and store final wrap–aware $Z_p$ . . . . . 63
5.10 FindBestByZp: return index of ant with maximal final $Z_p$ . . . . . 64

# Chapter 1

# Introduction

In scientific computing, evaluating mathematical derivatives is fundamental, as it is often used in key algorithms for simulation, optimization, and the numerical solution of differential equations. To solve nonlinear systems or nonlinear least-squares problems, prominent methods like Newton's method and its other alterations need the repeated evaluation of the Jacobian matrix and the solution of associated linear systems at each iteration. Due to the advancement of modern computing power and efficient software systems, the demand for solving complex and large problems is also growing in parallel.

Fortunately, observing the real-life large-scale problems, in most cases, we see that the problem exhibits sparsity, where Jacobian matrices contain a significant proportion of zero elements. Greater computational efficiency can be achieved if we leverage this sparsity, and the memory requirements can be reduced by avoiding storage of zeros. This accelerates computation by eliminating superfluous arithmetic operations and often leads to optimization of Jacobian calculation. Therefore, we can reduce processing cost if we know the sparsity structure of the Jacobian, as it does not change from iteration to iteration. Still, evaluating Jacobians for complex functions analytically is laborious and error-prone. The Finite-Difference (FD) method can be utilized to derive the Jacobian. While FD schemes offer a simple alternative, but introduce truncation errors. A more robust method is Algorithmic Differentiation (AD), which can compute derivatives to machine precision without truncation error.

Exploiting sparsity effectively for sparse matrices requires more complex and specialized algorithms that differ substantially from their dense-matrix counterparts. Performance for these algorithms is governed not only by floating-point operations but also by memory access patterns. The principle of data locality estimates that "recently accessed data (temporal) and nearby data (spatial) are likely to be accessed in the near future", is crucial [1]. Efficient algorithms minimize cache misses by organizing computations to maximize temporal and spatial locality, a key consideration for large-scale problems where data exceeds cache capacity.

In this thesis, we extend the work of Chandra et al. [2] and propose the novel (to our knowledge) approach of integrating Ant Colony Optimization (ACO) Heuristics to solve the Consecutive Zeros Problem (CZP), which is an extension of the well-known maximum consecutive ones problem. We introduce two variants of ACO, $Z_p$-Hybrid and $Z_p$-Estimator, to solve the CZP efficiently. Moreover, we define Greedy Heuristics for the problem, which exploit the existing structure of the Jacobian and try to improve the solution for CZP. We discuss in detail our contribution in this chapter in section 1.2. Before that, it is practical to elaborate on relevant research works done so far in this field, which we organized in the next section. Finally, we conclude this chapter by discussing the organization of our work in this thesis.

## 1.1 Related Work

The optimization of sparse matrices and combinatorial problems has seen significant advancements through various methods and techniques. This research draws upon several foundational works in optimization and matrix theory.

Hossain (2002) [12] provides an extensive analysis of direct and substitution methods

for determining sparse Jacobian matrices. The authors focused on reducing the number of automatic differentiation (AD) passes. Direct methods, like the CPR method, determine non-zeros directly from compression products. They require several groups $p$ for the compression. Substitution methods exploit a more nuanced sparsity structure by requiring that each row of the compressed Jacobian matrix contains a sequence of consecutive zeros. The minimum of all the sequences of consecutive zeros, $d$, allows for the construction of a seed matrix with only $p-d$ columns. Consequently, the method saves an additional $d$ number of AD passes, achieving significant computational savings over direct determination, and the authors further propose column rearrangement heuristics to actively maximize this value of $d$ and enhance the efficiency of the substitution scheme.

Hasan et al. (2016) [9] provide the DSJM (Determine Sparse Jacobian Matrices) toolkit, which represents a significant contribution to the efficient determination of sparse Jacobian matrices. The toolkit is designed using a "pattern graph" framework and employs cache-friendly array-based sparse data structures. Key features include a greedy grouping algorithm and several ordering heuristics that improve computational efficiency. Their numerical tests show that DSJM outperforms similar software in terms of timing and partitioning quality, highlighting the practical benefits of advanced sparse matrix handling techniques for large-scale problems.

Chandra and Hossain (2013) [2] present a method for determining sparse Jacobian matrices using substitution methods. Given that the sparsity pattern of the matrix is known, they incorporate graph coloring techniques to determine a compression scheme. They aim to align zero entries consecutively by column reordering of the compressed pattern matrix, framing this task as a variant of the TSP, where the distance cost is calculated for each pair of columns in the matrix. Their results suggest that nearest-neighbor heuristic algorithms perform reasonably well in this context, providing a foundation for integrating sparse ma-

trix considerations into optimization frameworks.

Hossain and Khan (2018) [10] introduce a detailed examination of exact methods for coloring sparse matrices. For a sparse matrix, ensuring that no two columns in the same group share nonzero entries in the same row position, they address the challenge of partitioning matrix columns into the fewest possible groups. Their study introduces efficient sparse data structures, emphasizes the use of branch-and-bound approaches, and proposes a novel tie-breaking method. This work highlights the importance of exact methods in achieving optimal coloring solutions and demonstrates the impact of advanced algorithms on computational efficiency and coloring quality. Their findings are particularly relevant for our approach.

Dorigo and Stützle (2004) [7] describe a comprehensive exploration of Ant Colony Optimization (`ACO`), a metaheuristic inspired by the foraging behavior of ants. Their book Ant Colony Optimization outlines the key principles and mechanisms of `ACO`, including pheromone updating and probabilistic decision-making. They demonstrate `ACO`'s versatility and effectiveness in solving complex combinatorial problems, such as the Traveling Salesperson Problem (TSP), scheduling, and network design. The authors highlight various adaptations of the `ACO` algorithm, showcasing its practical applications and empirical success. Their foundational work supports our research.

## 1.2 Our contribution

The summary of our work and research findings is outlined below:

- Study existing research works on efficient Jacobian matrix determination and the integration of `CZP`.
- Thorough analysis of the `ACO` heuristic and its data structure and implementation.

- Frame the CZP as a variant of TSP and implement ACO that aligns with respect to our goal.
- We introduce two variants of ACO, $Z_p$-Hybrid and $Z_p$-Estimator, which showcase Zp-oriented design to provide improved results on CZP. $Z_p$ is the minimum of the maximum length of consecutive zeros (column wrap-around) for each row of the matrix. This number is crucial as an increased number of $Z_p$ is utilized to compress the matrix further using the substitution method [12].
- Greedy Heuristics is proposed to exploit the structure of the matrix and run iteratively to obtain results to improve the solution for CZP. We define three variants of Greedy Heuristics: Exhaustive Approach, Selective Exhaustive, and Single Attempt.
- Comparative performance profiling for our results with Nearest-neighbor heuristic and 2-opt from [2].
- Constructed matrices were crafted based on the properties of real-world counterparts, and used to highlight the superiority of baseline ACO on these matrices.
- For the square matrices utilized in our implementation, we use a cost array (section 4.2) to map the matrix to an array, reducing the dimensionality and space complexity of the data structure.
- Extensive evaluation of the proposed methodologies on a wide range of large sparse matrices from [16].

## 1.3 Thesis Organization

We arrange this thesis into seven chapters. The first chapter is the introductory chapter, establishing the problem we are solving and the significance of the problem in a real-life scenario. This chapter also discusses the necessary literature work pertinent to our research. Finally, we discussed the contribution of our research work in solving the problem.

Chapter 2 provides the scientific background and preliminaries required for an in-depth understanding of the problem we are solving in this work. The chapter concludes with Implementation Process (section 2.9), describing the workflow sequentially, incorporating all the background knowledge discussed.

In Chapter 3, we illustrate the procedure of Greedy Heuristics and its variants with associated algorithms. We describe our implementation of ACO in Chapter 4, where we focus on the modifications required to frame the ACO with respect to our problem. The chapter is supported with relevant examples and algorithms. In Chapter 5, we introduce the structure and workflow of two Zp-aware ACO variants, $Z_p$-Hybrid and $Z_p$-Estimator, and analyze the factors that empower the heuristics to provide improved results. Chapter 6 consists of results and analysis for the test cases used in our work for the proposed methodologies, and we conclude our thesis with the Conclusion and Future Works in Chapter 7.

# Chapter 2

# Background and Methodology

We discuss thoroughly the relevant background and preliminaries to our research in this chapter. We aim to determine the sparse Jacobian matrices efficiently. To establish the necessary context for the problem, we start by introducing the `sparse Jacobian matrix`. Following, we discuss the significance of determining Jacobian through the renowned, widely used `Newton's Method`, and then the section `Direct Determination` explains the process of evaluating a Jacobian. Subsequently, we explain the `Compressed Sparse Row (CSR)` and its implementation in our work. In `Matrix Partitioning and Seed Matrix Computation`, we exhibit the process of a compression scheme on the Jacobian using coloring/partitioning information using DSJM [9]. We explain thoroughly the process of `Substitution methods` and the section `Consecutive Zeros Problem` provides an in-depth analysis of a strategy adapted to increase the efficiency of `Substitution methods`, which facilitates faster computation of the Jacobian. Finally, we illustrate the entire process of efficiently reconstructing the Jacobian, combining all the prior-discussed backgrounds in `Implementation Process`.

## 2.1 Sparse Matrix

In scientific computing, many matrices arise that are composed of a large number of zeros. These matrices are called sparse matrices, where the number of non-zeros, denoted as *nnz*, is much smaller than the number of entries (row $\times$ column), and the computational advantages can be gained from the knowledge of zero entries. Figure 2.1 displays a struc-

tural plot of a sparse matrix, *bcsstm*26, with 1922 rows and 1922 columns. The matrix has only 1992 non-zero entries out of 3694084 $(1922 \times 1922)$ total entries. The matrix has 99.95% zero entries; therefore, we can say the sparsity of the matrix is 99.95%.

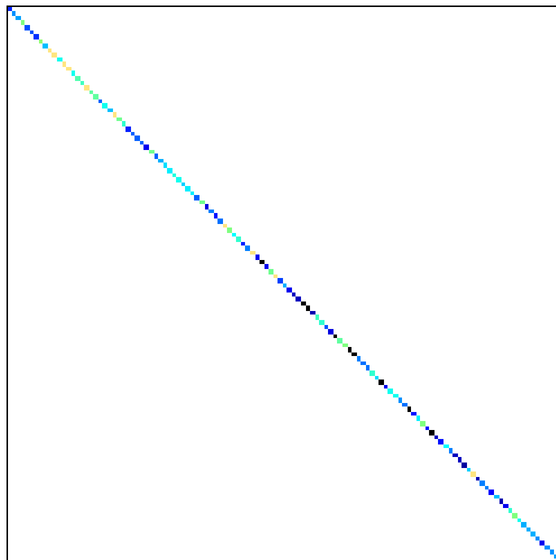


Figure 2.1: Structure plot of sparse matrix bcsstm26 (Dimensions: 1922 × 1922, 1922 non-zero entires). Source [4]

## 2.2 Jacobian Matrix

Let a vector-valued function $F : \mathbb{R}^n \to \mathbb{R}^m$ be continuously differentiable with $F(x) = \left(f_1(x), f_2(x), \ldots, f_m(x)\right)^{\mathsf{T}}$ and $x = (x_1, \ldots, x_n)^{\mathsf{T}} \in \mathbb{R}^n$. The Jacobian matrix $J(x)$ of $F$ at $x$ is the $m \times n$ matrix of first-order partial derivatives

$$J(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} \tag{2.1}$$

Equivalently, $J_{ij}(x) = \partial f_i/\partial x_j$, and $J(x)$ represents the differential or total first order derivative of $F$ at $x$.

## 2.3 Newton's Method

In solving optimization problems, Newton's method is one of the most effective techniques widely used. We consider solving a system of nonlinear equations,

$$\text{find } x^\star \in \mathbb{R}^n \quad \text{such that,} \quad F(x^\star) = 0, \qquad F : \mathbb{R}^n \to \mathbb{R}^m.$$

where $F$ is continuously differentiable, for $F$ at the current iterate $x^{(k)}$ first order Taylor approximation,

$$F(x^{(k)} + s) \approx F(x^{(k)}) + F'(x^{(k)})s$$

We replace $F'x^{(k)}$ as Jacobian $J(x^{(k)})$

$$F(x^{(k)} + s) \;\approx\; F(x^{(k)}) + J(x^{(k)})\,s,$$

so the *Newton step* $s^{(k)}$ solves

$$J(x^{(k)})\,s^{(k)} \;=\; -F(x^{(k)}),$$

Each iteration, we get an update of $s^{(k)}$ for $x$, getting closer to the root,

$$x^{(k+1)} = x^{(k)} + s^{(k)}$$

If there exist $F(x^\star) = 0$ for $x^\star \in \mathbb{R}^n$, then with a nonsingular $J(x^\star)$ and a good initial guess, Newton's method converges *quadratically* [15]; however, as every iteration requires solving a linear system as well as forming the Jacobian or at least multiplication operation of Jacobian, therefore, for sparse large problems, the computational cost is dominated by

Algorithm 2.1: Newton's method, source [15]

NEWTONMETHOD$(F, x^{(0)})$
1 **repeat** Compute $J(x^{(k)})$
2 Solve $J(x^{(k)})\, s^{(k)} = -F(x^{(k)})$
3 Set $x^{(k+1)} = x^{(k)} + s^{(k)}$
4 Set $k = k+1$
5 **until** $x^{(k)}$ is a "good enough" approximation to $x^{\star}$

it. Hossain et al. (2013) [15] provide an in-depth analysis of Newton's method, explaining it in detail.

For the Jacobian, two methods are utilized: (i) finite differences (FD) of directional derivatives,

$$J(x)\, s \;\approx\; \frac{F(x+\varepsilon s) - F(x)}{\varepsilon},$$

And (ii) forward-mode algorithmic differentiation (AD) (AD avoids FD step-size tuning, truncation, and cancellation error [15]) computes $J(x)s$ to machine precision. We exploit the *structural orthogonality* of matrix, where columns $j$ and $\ell$ are placed together in the same group, given that no row in both columns contains non-zeros in the same row location as illustrated in the process of seed matrix computation in section 2.6; We can therefore, combine all the structural orthogonality columns of a group into a single column, as a result, one pass in direction $s = e_j + e_\ell + \cdots$ recovers all columns in the group. This significantly reduces the number of FD/AD passes. If $S$ (the *seed matrix*) collects these group directions, then we define

$$B \;=\; J(x)\, S,$$

Subsequently, we can read off the original columns from the compressed columns of $B$, following the idea explained by Curtis et al. [3].

## 2.4 Direct Determination

To solve the nonlinear optimization problems and differential equations, derivatives are needed. Let $F : \mathbb{R}^n \to \mathbb{R}^m$ be the once continuously differentiable function mapping and a point $x \in \mathbb{R}^n$, we consider determining the Jacobian $F'(x)$. For a direction $s \in \mathbb{R}^n$, the directional derivative may be approximated by a forward difference ([14]):

$$\frac{\partial}{\partial t} F(x+ts)\Big|_{t=0} = F'(x)s \approx As = \frac{1}{\varepsilon}\big[F(x+\varepsilon s) - F(x)\big] \equiv b,$$

using one additional evaluation of $F$ at $x+\varepsilon s$ (with $\varepsilon > 0$ small). In forward-mode Algorithmic Differentiation (AD), the product $b = F'(x)s$ can be computed to machine accuracy at a cost that is a small multiple of a single function evaluation. The *Jacobian matrix determination problem* (JMDP) is therefore [14]: choose directions $s_j \in \mathbb{R}^n$, $j = 1, \dots, p$, so that from the matrix–vector products $b_j = As_j$ (or equivalently, $B = AS$ with $S = [s_1 \; \cdots \; s_p]$), where the Jacobian $A = F'(x)$ is uniquely determined.

Two columns $A(:,j)$ and $A(:,\ell)$ are called *structurally orthogonal* if they never have nonzeros in the same row. We write $A(:,j) \perp A(:,\ell)$ when there is no $i$ with $a_{ij} \neq 0$ and $a_{i\ell} \neq 0$ (otherwise $A(:,j) \not\perp A(:,\ell)$). If $A(:,j) \perp A(:,\ell)$, a single additional evaluation with the combined direction $s = e_j + e_\ell$ yields the sum of those two columns directly:

$$F'_j + F'_\ell = A(:,j) + A(:,\ell) = \frac{1}{\varepsilon}\Big(F\big(x+\varepsilon(e_j+e_\ell)\big) - F(x)\Big).$$

This observation underlies grouping columns into structurally orthogonal sets so several columns can be recovered per function (or AD) evaluation.

## 2.5 Compressed Sparse Row (CSR) Format

Compressed Sparse Row (CSR) is a storage scheme where we can save the sparse matrix using only three arrays. CSR utilizes the sparsity of the matrix and only stores the nonzeros of the matrix. To locate the the nonzeros of $A \in \mathbb{R}^{m \times n}$, CSR uses three arrays: `val` $\in$

$\mathbb{R}^{\texttt{nnz}}$, $\texttt{colIdx} \in \{0,\ldots,n-1\}^{\texttt{nnz}}$, and $\texttt{rowPtr} \in \{0,\ldots,\texttt{nnz}\}^{m+1}$. In colIdx we record the column index of each nonzero; corresponding matrix values for the columns are stored in val in the same index order; rowPtr[i] is the starting offset in val/colIdx for row $i$, with $\texttt{rowPtr}[m] = \texttt{nnz}$. The nonzeros of row $i$ occupy the contiguous slice from $\texttt{rowPtr}[i]$ (including) to $\texttt{rowPtr}[i+1]$ (excluding),

$$k \in \big[\texttt{rowPtr}[i], \texttt{rowPtr}[i{+}1]\big),$$

For any value of $k$ for row $i$, $\texttt{val}[k]$ and $\texttt{colIdx}[k]$ represent the value and column number of the entry in the matrix. Thus, CSR enables rowwise access by reading colIdx and val between $\texttt{rowPtr}[i]$ and $\texttt{rowPtr}[i{+}1]$. It requires $2\,\texttt{nnz}+m+1$ scalar slots (val and colIdx of length nnz, and rowPtr of length $m+1$). This Contiguous placement of these arrays in memory improves cache efficiency and spatial locality.

**Example** Let

$$A = \begin{pmatrix} a_{11} & 0 & 0 & a_{14} & 0 & 0 \\ 0 & a_{22} & 0 & 0 & a_{25} & a_{26} \\ 0 & 0 & a_{33} & 0 & 0 & 0 \\ a_{41} & 0 & 0 & a_{44} & 0 & 0 \\ 0 & a_{52} & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Here, the CSR encoding for the matrix is:

$$\texttt{val} = [a_{11}, a_{14},\ a_{22}, a_{25}, a_{26},\ a_{33},\ a_{41}, a_{44},\ a_{52}], \quad \texttt{colIdx} = [0,3,\ 1,4,5,\ 2,\ 0,3,\ 1],$$

$$\texttt{rowPtr} = [0,2,5,6,8,9],$$

so that row–slice mapping is

$row_0 \mapsto [0,2)$(from 0 to 1, excluding 2),

$row_1 \mapsto [2,5)$(from 2 to 4, excluding 5),

$row_2 \mapsto [5,6)$(from 5 to 5, excluding 6),

$row_3 \mapsto [6,8)$(from 6 to 7, excluding 8),

$row_4 \mapsto [8,9)$(from 8 to 8, excluding 9),

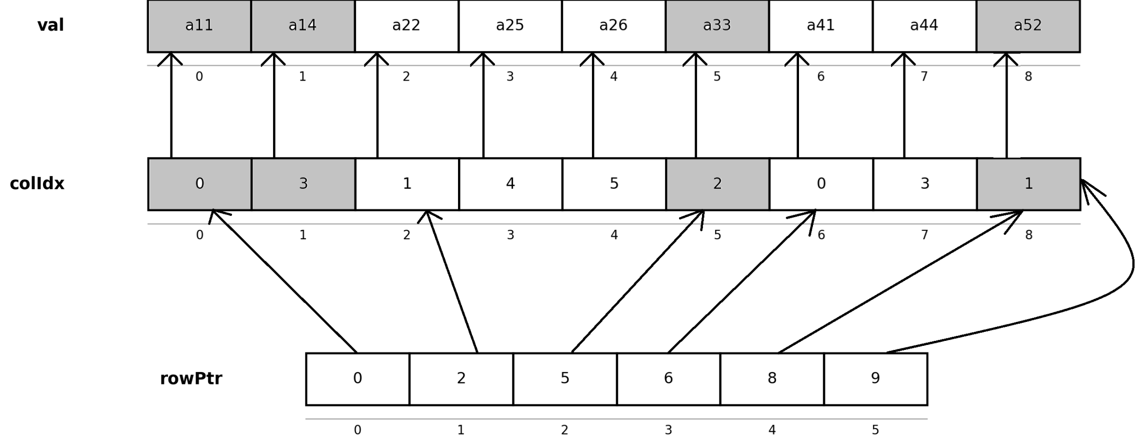



Figure 2.2: CSR arrays: `val`, `colIdx`, `rowPtr`.

## 2.6 Matrix Partitioning and Seed Matrix Computation

Without any utilization of the sparsity pattern of a Jacobian $A \in \mathbb{R}^{m \times n}$, evaluation of the matrix requires $n$ number of FD or AD passes to recover $A$ column by column. In contrast, as discussed in section 2.4, if we partitioned $A$ and group the `structurally orthogonal` columns together to $n_c$ different groups, then the column size is reduced from $n$ to $n_c$, the FD/AD passes are reduced by $n - n_c$. In this work, we used the state-of-the-art coloring algorithms from DSJM [9] to get partition/coloring information for the Jacobin matrix. From the partition information, we used the best result to create the *seed matrix* $S \in \{0,1\}^{n \times n_c}$. The seed matrix is used to compress matrix $A$ into $B \in \mathbb{R}^{m \times n_c}$.

$$B = AS = \left[ As^{(1)} \; As^{(2)} \; \cdots \; As^{(n_c)} \right],$$

As columns within each group are `structurally orthogonal`, each $As^{(g)}, g \in n_c$ is just a compressed column of $A$, so the original columns can be read off directly. As a result, we can scan the original columns directly exploiting sparsity via column partitioning, as explained in [3].

We considering the $5 \times 6$ sparse matrix from section 2.5

$$A = \begin{pmatrix} a_{11} & 0 & 0 & a_{14} & 0 & 0 \\ 0 & a_{22} & 0 & 0 & a_{25} & a_{26} \\ 0 & 0 & a_{33} & 0 & 0 & 0 \\ a_{41} & 0 & 0 & a_{44} & 0 & 0 \\ 0 & a_{52} & 0 & 0 & 0 & 0 \end{pmatrix}.$$

The row positions of the non-zeros of columns are (considering the starting column index is 1):

$$\text{nnz}(A(:,1)) = \{1,4\}, \quad \text{nnz}(A(:,2)) = \{2,5\}, \quad \text{nnz}(A(:,3)) = \{3\},$$
$$\text{nnz}(A(:,4)) = \{1,4\}, \quad \text{nnz}(A(:,5)) = \{2\}, \quad \text{nnz}(A(:,6)) = \{2\}.$$

The obtained non-orthogonal pairs for $A$ are $(1,4)$, $(2,5)$, $(2,6)$, and $(5,6)$. Partitioning the columns into structurally orthogonal groups, we can get three groups, such as,

$$\mathcal{G}_1 = \{1,3,5\}, \qquad \mathcal{G}_2 = \{2,4\}, \qquad \mathcal{G}_3 = \{6\}.$$

The corresponding seed vectors are

$$s^{(1)} = C_1 + C_3 + C_5, \qquad s^{(2)} = C_2 + C_4, \qquad s^{(3)} = C_6,$$

and the seed matrix $S = [s^{(1)}\ s^{(2)}\ s^{(3)}] \in \{0,1\}^{6\times 3}$ is

$$S = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Calculating $B = AS$ yields a $5 \times 3$ matrix as below:

$$B = AS = \big[A(:,1)+A(:,3)+A(:,5)\ \ A(:,2)+A(:,4)\ \ A(:,6)\big] = \begin{pmatrix} a_{11} & a_{14} & 0 \\ a_{25} & a_{22} & a_{26} \\ a_{33} & 0 & 0 \\ a_{41} & a_{44} & 0 \\ 0 & a_{52} & 0 \end{pmatrix}.$$

Following the formation of the compressed matrix $B$, three products $As^{(g)}$, equivalently, three FD/AD passes suffice to recover all six columns of $A$, instead of six, if we differentiate one column at a time.

With forward differences, each group product is approximated by

$$As^{(g)} \approx \frac{1}{\varepsilon}\Big(F\big(x+\varepsilon s^{(g)}\big) - F(x)\Big), \quad g = 1,2,3,$$

while forward-mode AD computes $As^{(g)}$ to roundoff accuracy at comparable cost. The ultimate goal is to *minimize* the number of `structurally orthogonal` groups, hence minimizing evaluations. However, finding a minimum such partition is equivalent to a vertex-coloring problem on the column-intersection graph of $A$, which is an NP-hard prob-

Algorithm 2.2: CSR Matrix Multiplication

```
CSR-MATRIX-MULTIPLY(valuesA, col_indexA, row_ptrA, valuesB, col_indexB, row_ptrB)
 1  rowsA = row_ptrA.size() − 1
 2  row_ptrC = new vector of size rowsA + 1 initialized to 0
 3  tempC = new vector of rowsA unordered maps
 4  for i = 0 to rowsA − 1
 5      for j = row_ptrA[i] to row_ptrA[i + 1] − 1
 6          colA = col_indexA[j]
 7          valA = valuesA[j]
 8          for k = row_ptrB[colA] to row_ptrB[colA + 1] − 1
 9              colB = col_indexB[k]
10              valB = valuesB[k]
11              tempC[i][colB] = tempC[i][colB] + valA × valB
12  for i = 0 to rowsA − 1
13      row_ptrC[i + 1] = row_ptrC[i] + tempC[i].size()
14  valuesC.reserve(row_ptrC[rowsA])
15  col_indexC.reserve(row_ptrC[rowsA])
16  for i = 0 to rowsA − 1
17      sortedCols = sort tempC[i] by column index
18      for each (col, val) in sortedCols
19          col_indexC.push_back(col)
20          valuesC.push_back(val)
21  return (valuesC, col_indexC, row_ptrC)
```

lem; therefore, heuristics are used to compute good colorings and the corresponding seed matrices. In our implementation, we employed the well-defined implementation of coloring heuristics from DSJM [9]. We stored binary matrix $A$ and seed matrix $S$, in CSR format described in section 2.5. Then, we calculated binary $B$, CSR matrix multiplication of $A$ and $S$, as illustrated in algorithm 2.2. The algorithm takes $valuesA, col_indexA, row_ptrA$ for binary matrix $A$ and $valuesB, col_indexB, row_ptrB$ for seed matrix $S$ as an input, and generates $valuesC, col_indexC, row_ptrC$ representing the binary matrix $B$.

## 2.7 Substitution methods

While estimating a sparse Jacobian $A \in \mathbb{R}^{m \times n}$ with finite differences (FD) or automatic differentiation (AD), we follow the strategy of seeding multiple *structurally orthogonal*

columns at once and then recover individual Jacobian entries from a small linear system. As illustrated in section 2.6, let $S \in \mathbb{R}^{n \times n_c}$ be a seed matrix that encodes a partition of the columns of $A$ into $n_c$ groups, and form

$$B \leftarrow AS \quad \text{(one forward AD pass required per column of } S\text{),} \tag{2.2}$$

We aim to minimize the number of AD passes further. Hossain et al. [11], [12] introduce the substitution method employed to compress the Jacobian further based on $d$ of the matrix, where $d$ is the smallest of the maximum consecutive zero counts (column wrap-around) for each row in the matrix.

Following is an example of a matrix having $d = 2$, where the number of maximum consecutive zeros is 3 in the first row and 2 in every other row. We consider column wrap-around; therefore, the leading and trailing zeros in the fourth row are considered as maximum consecutive zeros $= 2$.

$$\begin{matrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{matrix}$$

Let $Z_i$ be the maximum number of consecutive zeros with column wrap-around in row $i$ of $B \in \mathbb{R}^{m \times n_c}$, and $Z_p$ be the minimum of all $Z_i$ in the matrix. Therefore, here we denote $d$ as $Z_p$. Let the columns be partitioned into $n_c - Z_p$ groups $\widetilde{C}_1, \ldots, \widetilde{C}_{n_c - Z_p}$. Given that $n_c > Z_i$ and $Z_p >= 1$, we can merge adjacent groups in the seed to compress the Jacobian further, reducing the number of AD passes by $Z_p$.

### 2.7.1 Basic two–group substitution

Here, $Z_p = 1$, the merging of columns for two-group substitution to obtain $\widetilde{B}$ from $B$:

$$\widetilde{C}_1 = \{C_1, C_2\},\ \widetilde{C}_2 = \{C_2, C_3\},\ \ldots,\ \widetilde{C}_{n_c-1} = \{C_{n_c-1}, C_{n_c}\}. \tag{2.3}$$

The seed matrix, $S_{conz} \in \{1,0\}^{n_c \times n_c - Z_p}$ to obtain the above arrangement using $Z_p = 1$:

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 0 & \ddots & \vdots \\ 0 & 1 & 1 & \cdots & 0 \\ 0 & 0 & 1 & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

The seed matrix entries are determined by formula below, $S_{conz_{i,j}}$ (row $i$, column $j$) is given by:

$$S_{conz_{i,j}} = \begin{cases} 1 & \text{if } \max(1, i - Z_p) \leq j \leq i \\ 0 & \text{otherwise} \end{cases} \quad \text{for } i = 1, 2, \ldots, n_c, \quad j = 1, 2, \ldots, n_c - Z_p. \tag{2.5}$$

Using the $S_{conz}$ for $Z_p = 1$, we get $\widetilde{B}$,

$$
B \times S_{\text{conz}} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n_c} \\ a_{21} & a_{22} & \cdots & a_{2n_c} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn_c} \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 0 & \ddots & \vdots \\ 0 & 1 & 1 & \cdots & 0 \\ 0 & 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & 0 & 1 \end{bmatrix}
$$

$$
= \begin{bmatrix} (a_{11}+a_{12}) & (a_{12}+a_{13}) & \cdots & (a_{1,n_c-1}+a_{1,n_c}) \\ (a_{21}+a_{22}) & (a_{22}+a_{23}) & \cdots & (a_{2,n_c-1}+a_{2,n_c}) \\ \vdots & \vdots & \ddots & \vdots \\ (a_{m1}+a_{m2}) & (a_{m2}+a_{m3}) & \cdots & (a_{m,n_c-1}+a_{m,n_c}) \end{bmatrix} \tag{2.6}
$$

This compression scheme ensures $n_c - Z_p$ number of FD/AD passes instead of $n_c$ passes, significantly reducing the computation time for large sparse matrices. Following the computation, we can recover $B$ from $\widetilde{B}$ using the substitution method. Let $\alpha_1, \ldots, \alpha_{n_c}$ denote the (at most $Z_i$) unknown nonzeros in a fixed row $i$ of $A$, ordered consistently with $C_1, \ldots, C_{n_c}$, and let $\beta_1, \ldots, \beta_{n_c-1}$ be the corresponding entries of row $i$ in $B$. The row system produced by (2.3) has the bidiagonal form

$$
\begin{bmatrix} 1 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 1 & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_p \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{p-1} \end{bmatrix}. \tag{2.7}
$$

Because at most $Z_i < n_c$ of the $\alpha_k$ are nonzero, at least one $\alpha_k$ is known to be zero, which makes (2.7) triangular after deleting that variable and permits recovery by substitution. Consequently, *two–group substitution* saves at least one AD pass when $n_c > Z_i$ (precise statement and proof as Lemma 1 in [11]).

### 2.7.2 Generalized $Z_p$–group substitution

We can save further computational cost if the compressed matrix rows have greater $Z_i$ values. Let every row of the compressed matrix have at least $Zp$ consecutive zeros. Therefore, the columns can be merged and compressed to $n_c - Z_p$ columns, where each column is a group of $Z_p+1$ columns,

$$\widetilde{C}_1 = \{C_1,\dots,C_{Z_p+1}\},\ \widetilde{C}_2 = \{C_2,\dots,C_{Z_p+2}\},\ \dots,\ \widetilde{C}_{n_c-Z_p} = \{C_{n_c-Z_p},\dots,C_{n_c}\} \tag{2.8}$$

Then we obtain a block triangular structure per row, same as subsection 2.7.1. The substitution method guarantees the recovery of all the non-zeros, while using only $n_c - Z_p$ AD passes. This is formalized as Lemma 2 in [11].

### 2.7.3 Illustrative example

We display the well-defined example from [11]. Let

$$A = \begin{bmatrix} a_{11} & 0 & a_{13} \\ a_{21} & a_{22} & 0 \\ 0 & a_{32} & a_{33} \end{bmatrix}, \qquad X = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix},$$

so $B = AX = \begin{bmatrix} b^{(1)} & b^{(2)} \end{bmatrix}$ comes from two forward AD passes (merging groups $b^{(1)} = \{Column_1, Column_2\}$ and $b^{(2)} = \{Column_2, Column_3\}$ of $A$). For the second row,

$$\begin{bmatrix} a_{21} & a_{22} & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} b_{21} & b_{22} \end{bmatrix}.$$

Deleting the known zero row (third row of $X$) and transposing yields the $2\times 2$ upper–triangular system

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a_{21} \\ a_{22} \end{bmatrix} = \begin{bmatrix} b_{21} \\ b_{22} \end{bmatrix},$$

so $a_{22} = b_{22}$ and $a_{21} = b_{21} - a_{22}$. Thus, any direct determination would require 3 additional function evaluations; however, in this case, with the substitution method, it can be evaluated using only two additional function evaluations.

## 2.8 Consecutive Zeros Problem

In a sparse matrix, we can observe a large number of zeros compared to non-zero values. Minimizing the distance between adjacent columns [2] by permuting the columns results in a long sequence of consecutive zeros in each row of the matrix. Denote by $A_\Pi \equiv A \times I_\Pi$ where $I_\Pi$ is the permutation matrix representing permutation $\Pi$. The combinatorial optimization problem that we aim to solve is,

> **Consecutive Zeros Problem (CZP):** Given the sparsity pattern of a matrix $A \in \mathbb{R}^{m\times n}$ find a permutation $\Pi$ of the columns of $A$ that maximizes
>
> $$Z_p = \min_i \{Z_i\}$$
>
> where $Z_i$ is the maximal sequence of consecutive zero entries (with column wrap-around) in row $i$ of $A_\Pi, i = 1, 2, \ldots, n$.

The central idea of our heuristic method is to represent the matrix ordering problem as a variant of the Traveling Salesperson Problem (TSP). For every pair of columns $j, l$ of matrix $A$, we define cost

$$c_{jl} = \sum_{i=1}^{n} A(i,j) \odot A(i,l),$$

where,

$$A(i,j) \odot A(i,l) = \begin{cases} 0, \text{ if } (A(i,j) = A(i,l) = 0) \text{ or } (A(i,j) \neq 0 \text{ and } A(i,l) \neq 0) \\ 1, \text{ otherwise} \end{cases}.$$

Our objective is to find a permutation $\Pi$ of the columns that minimizes the cost $C(A_\Pi) = \sum_{j=1}^{n-1} c_{\Pi(j)\Pi(j+1)}$, therefore maximizing $Z_p$. The following is the illustration of the cost calculation of *ColumnA* and *ColumnB*.

| COLUMN A | COLUMN B | |
|---|---|---|
| 1 | 0 | +1 |
| 0 | 0 | +0 |
| 1 | 1 | +0 |
| 0 | 1 | +1 |
| 1 | 1 | +0 |
| 1 | 0 | +1 |
| 0 | 1 | +1 |
| 0 | 0 | +0 |
| 1 | 1 | +0 |
| 0 | 1 | +1 |
| | COST = | 5 |

Parameters related to the Consecutive Zeros problem are:

$\mathbf{Z}_{min}$: Minimum number of zeros in any row across the matrix.

$\mathbf{Z}_{max}$: Maximum number of zeros in any row across the matrix.

$\mathbf{Z}_i$: Length of maximum consecutive zeros in row i, considering the column wrap-around.

$\mathbf{Z}_t$: Total number of zeros in a row

$\mathbf{Z}_p$: Minimum of all the $Z_i$ in the matrix

## 2.9 Implementation Process

In this section, we explain the process of efficiently reconstructing a Jacobian matrix, incorporating all the background established in this chapter. The workflow of the implementation is arranged sequentially in algorithm 2.3. Our goal is to minimize the AD/FD passes while determining the Jacobian $A \in \mathbb{R}^{m\times n}$, derived from a continuously differentiable func-

Algorithm 2.3: Efficient Reconstruction of Jacobian Matrix

DETERMINE-SPARSE-JACOBIAN($A$)
1 **Input:** Sparse matrix $A$
2 **Output:** Reconstructed sparse Jacobian matrix
3 $A_{\text{binary}} = \text{SPARSITY-PATTERN}(A)$
4 $S_{\text{binary}} = \text{COLOR-COLUMNS}(A_{\text{binary}})$
5 $B_{\text{binary}} = A_{\text{binary}} \times S_{\text{binary}}$
6 $B_{\text{TSP}} = \text{CONVERT-TO-TSP}(B_{\text{binary}})$
7 Perm, $Z_p = \text{GREEDY/ANT-COLONY-HEURISTICS}(B_{\text{TSP}})$
8 $S_{\text{binary}} = S_{\text{binary}} \times \text{Perm}$ // Apply permutation
9 $S_{\text{conz}} = SUB_S(Z_p)$ // Create substitution seed for Zp consecutive zeros
10 $S_{\text{binary}} = S_{\text{binary}} \times S_{\text{conz}}$
11 $B = \text{EVALUATE-ADFD}(F, x, S_{\text{binary}})$ // Evaluate using AD into B
12 SOLVE-FOR-A$(A, S_{\text{binary}}, B)$ // Solve m triangular linear equations to recover the Jacobian
13 **return** $A$

tion, $F : \mathbb{R}^n \to \mathbb{R}^m$ and variables $x \in \mathbb{R}^n$. From the sparsity information of the Jacobian matrix $A$, we form $A_{binary} \in \{0,1\}^{m\times n}$ and, using $A_{binary}$, a seed matrix, $S_{binary} \in \{0,1\}^{n\times n_c}$ is created employing the coloring/partitioning algorithm from [9]. We get $B_{binary} \in \{0,1\}^{m\times n_c}$ as product of matrix multiplication of $A_{binary}$ and $S_{binary}$. We frame the $B_{binary}$ as a TSP problem using the concepts from section 2.8, [2], and employ one of the proposed heuristics in this thesis (`Greedy Heuristic`/ `Ant Colony Optimization Heuristic`/ `Zp-Hybrid Heuristic`/ `Zp-Estimator Heuristic`). The heuristics efficiently improve the $Zp$ in most cases for the Jacobian and provide the column permutation *Perm* and improved $Z_p$. Now using the *Perm*, we can rearrange $S_{binary}$ columns, and the $Z_p$ information is utilized in the substitution method [12], generating the seed matrix, $S_{conz} \in \{0,1\}^{n_c\times(n_c-Z_p)}$. Permuted $S_{binary}$ and derived $S_{conz}$, both utilized to form $S_{binary}$, which is taken as input with function $F$ and variables $x$ to determine the Jacobian $A$ with the help of AD/FD passes. As we have reduced the column from $n$ to $n_c$, then $n_c$ to $n_c - Z_p$, therefore, the determination of the Jacobian needs $n - n_c + Z_p$ fewer AD/FD passes. Using $A$, $S_{binary}$ and $B$ we can retrieve A.

### 2.9.1 Illustrative Example

Let matrix, $A \in \mathbb{R}^{5\times 8}$

$$A = \begin{bmatrix} 5 & 0 & 0 & 2 & 0 & 0 & 0 & 9 \\ 0 & 4 & 0 & 0 & 0 & 9 & 7 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 4 & 6 & 3 & 0 & 0 & 7 \\ 1 & 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

We can extract the sparsity pattern of the matrix $A$ and generate $A_{binary} \in \{1,0\}^{5\times 8}$

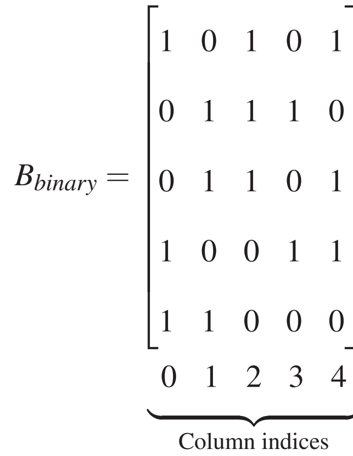
$$A_{\text{binary}} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare \end{bmatrix}$$

As we can see using DSJM [9], we can group the same colored columns into one single column. To obtain the column partitioning, the seed matrix, $S_{binary} \in \{1,0\}^{8\times 5}$ obtained

using DSJM,

$$S_{binary} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Now, we obtain $B_{binary} \in \{1,0\}^{5\times 5} = A_{binary} \times S_{binary}$




In order to utilize the substitution method [11], we need to obtain a permutation of $B$ that increases the $d/Z_p$ of the matrix. Therefore, we employ `Ant Colony Optimization/Greedy Heuristics/Zp-Hybrid ACO/Zp-Estimator ACO` and get a permutation for $B$.

$$B_{binary}(permuted) = \begin{matrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 2 & 4 & 0 & 3 & 1 \end{matrix}$$

Column indices

From the permuted $B$, we get $d = 2$, therefore, we can generate the Seed matrix [11], $S_{conz} \in \{1,0\}^{5\times3}$

$$S_{conz} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Now, we can update the $S_{binary} \in \{1,0\}^{8\times3} = S_{binary} \times S_{conz}$. Using the Seed matrices, we can reduce the column from 8 to 3. Using `AD/FD`, now we can evaluate the Jacobian with only 3 passes instead of 8. As we know the structure of $A$ and $B$ and the obtained seed matrix, $S_{binary}$, we can recover $A$.

# Chapter 3

# Greedy Heuristics

To improve the Zp of the Jacobian matrix $B \in \mathbb{R}^{m \times n_c}$, we introduce the Greedy Heuristics (GH), which prioritizes the rows constrained with the lower $Z_i$ in the matrix and iteratively tries to improve the $Z_i$ of the row by swapping columns. GH selects rows sequentially from the candidate row list formed using get_prioritized_candidates (algorithm 3.4) and identifies the longest consecutive zero block (lczb) with column wrap-around for the selected row using Find_Longest_Consecutive_Zeros function (algorithm 3.1). Depending on the GH variant, the heuristics performs adjacent column swaps around lczb to improve $Z_i$ for the row while ensuring the $Z_p$ of the matrix is not reduced. At any phase of the GH process, $Z_{min}$, $Z_p$, $Z_t$_*values*, $Z_i$_*values* are extracted using the evaluate_permutation (algorithm 3.2) and compute_zt_zi_permuted (algorithm 3.3) functions, requiring the current column permutation, col_permutation as input to the function. Here, $Z_{min}$ is the minimum number of 0's in any row of the matrix, $Z_t$_*values* and $Z_i$_*values* are arrays containing the total number of 0's and the longest length of consecutive 0's (column wrap-around) for each row in the matrix, respectively. We introduce three variants of the GH, namely the Exhaustive Approach, Selective Exhaustive, and Single Attempt. The function improve_Zp_permutation (algorithm 3.5) is the main improvement function, which builds the solution based on the choice of variant.

## 3.1 Improve Zp permutation

To improve the overall $Z_p$ of the matrix, we implement the improve_Zp_permutation

Algorithm 3.1: Find-Longest-Consecutive-Zeros: Locate maximum zero sequence with wrap-around

FIND-LONGEST-CONSECUTIVE-ZEROS(*row*, *permutation*)

```
1   n = row.size
2   max_len = 0, start_pos = 0, current_len = 0
3   for j = 0 to n − 1
4       col = permutation[j]
5       if row[col] = 0
6           current_len = current_len + 1
7           if current_len > max_len
8               max_len = current_len
9               start_pos = j − current_len + 1
10      else
11          current_len = 0
12  if row[permutation[0]] = 0 and row[permutation[n − 1]] = 0
13      left = 0
14      while left < n and row[permutation[left]] = 0
15          left = left + 1
16      right = n − 1
17      while right ≥ 0 and row[permutation[right]] = 0
18          right = right − 1
19      wrap_len = left + (n − 1 − right)
20      if wrap_len ≥ max_len
21          max_len = wrap_len
22          start_pos = (n − (wrap_len − left)) mod n
23  return (start_pos, max_len)
```

function. For each try, `improve_Zp_permutation` first constructs a list of candidate rows using the function `get_prioritized_candidates` (algorithm 3.4). As $Z_{min}$ is the maximum value that $Z_p$ can achieve for the matrix, rows with $Z_i \geq Z_{min}$ do not confine the $Z_p$ for the matrix, hence, are excluded from the candidate list. Additionally, rows with $Z_i \geq n_c - 1$ ($n_c$ is the number of columns) were also removed from the candidate list as having only one or no non-zero entries in the row have no significance in the improvement of $Z_p$ by column swap. Moreover, all the excluded rows from the candidate list are added to the `permanently_skipped` array while constructing the candidate list. Then, we introduce `priority score` to rank the remaining rows in the candidate list,

Algorithm 3.2: Evaluate-Permutation: Compute matrix metrics for given permutation

EVALUATE-PERMUTATION(*matrix*, *permutation*)

```
1  rows = matrix.size
2  Zmin = ∞, Zp = ∞
3  Initialize Zt_values[rows], Zi_values[rows]
4  for i = 0 to rows − 1
5      (Zt, Zi) = COMPUTE-ZT-ZI-PERMUTED(matrix[i], permutation)
6      Zt_values[i] = Zt
7      Zi_values[i] = Zi
8      Zmin = min(Zmin, Zt)
9      Zp = min(Zp, Zi)
10 return (Zmin, Zp, Zt_values, Zi_values)
```

where the constrained row gets a lower `priority score`. Using the `priority score`, `get_prioritized_candidates` prioritizes the rows with a lower score by placing the rows at the top of the candidate list to get processed before other rows. For the `priority score` calculation, we multiply the $Z_i$ of the row by 1000, aiming at increasing the score for rows with higher $Z_i$. In the case of multiple rows with the same $Z_i$ value, the row with a lower number of zeros is more constrained and should be prioritized. To achieve that goal, we add the difference of $Z_t$ and $Z_i$ of the row to the `priority score`. The `priority score` for the row $i$ is defined as,

$$priority_score = Z_i_value \times 1000 + (Z_t_value - Z_i_value) \tag{3.1}$$

After finalizing the candidate list, the function `improve_Zp_permutation` iterates through all the rows in the list, applies one of the variants of `GH (Exhaustive, Selective Exhaustive, Single Attempt)`, and adds the processed row to the `permanently_skipped` array. To identify the column wrap-around `lczb` for the selected row, all the variants use the function `Find_Longest_Consecutive_Zeros` (algorithm 3.1) and iteratively try to swap columns around the `lczb` to improve the $Z_i$ of the row. Upon getting any improvement of $Z_i$ without decreasing the $Z_p$ of the matrix, we obtain a new $Z_i$ for the row (possibly an improved $Z_p$

Algorithm 3.3: Compute-Zt-Zi-Permuted: Calculate total zeros and max consecutive zeros for a row

```
COMPUTE-ZT-ZI-PERMUTED(row, permutation)
 1  Zt = 0
 2  max_Zi = 0, current_Zi = 0, first_Zi = 0, last_Zi = 0
 3  first_segment = TRUE
 4  n = row.size
 5  for j = 0 to n − 1
 6      col = permutation[j]
 7      if row[col] = 0
 8          current_Zi = current_Zi + 1
 9          Zt = Zt + 1
10          if first_segment
11              first_Zi = first_Zi + 1
12      else
13          if current_Zi > 0
14              last_Zi = current_Zi
15              max_Zi = max(max_Zi, current_Zi)
16              current_Zi = 0
17          first_segment = FALSE
18  if current_Zi > 0
19      last_Zi = current_Zi
20      max_Zi = max(max_Zi, current_Zi)
21  if row[permutation[0]] = 0 and row[permutation[n − 1]] = 0
22      max_Zi = max(max_Zi, first_Zi + last_Zi)
23  if Zt = n
24      max_Zi = n
25  return (Zt, max_Zi)
```

for the matrix as well) and a new column permutation for the matrix. Consequently, with the new permutation, $Z_t$*_values*, $Z_i$*_values*, and *Z_min* for the matrix are updated. For the new permutation of the matrix, based on the updated values, we construct the candidate list anew, excluding the rows in `permanently_skipped` array, where the function repeats the process of evaluating rows and places them on the new candidate list based on the new `priority score`. If no improvement is found at any try, the function terminates, returning the corresponding $Z_p$ and permutation for the matrix.

Algorithm 3.4: Get-Prioritized-Candidates: Select rows for improvement prioritization

GET-PRIORITIZED-CANDIDATES($Zi_values, Zt_values, Zmin, permanently_skipped, n_c, outfile$)
1 Initialize *candidates* as empty list
2 **for** $i = 0$ **to** $Zi_values.size - 1$
3 **if** $Zi_values[i] \geq Zmin$ **Or** $Zi_values[i] \geq n_c - 1$
4 **continue**
5 **else** $priority_score = Zi_values[i] \times 1000 + (Zt_values[i] - Zi_values[i])$
6 $candidates.append((i, priority_score))$
7 Sort *candidates* by *priority_score* in ascending order
8 **return** *candidates*

## 3.2 Illustrative Example

Let

$$X = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}, [Z_t, Z_i] = \begin{bmatrix} 3,1 \\ 5,5 \\ 4,2 \\ 3,2 \end{bmatrix}$$

be a Jacobian matrix ($Z_t$ and column wrap-around $Z_i$ provided for each row), and our goal is to improve the $Z_p$ of the Jacobian matrix. We can observe, the $Z_{min}$ for the matrix is 3 and $Z_p$ is 1; therefore, constructing a candidate list for rows, we can exclude row 2 ($Z_i \geq Z_{min}$) and save it in `permanently_skipped`. The `priority_scores` for the remaining rows are:

$$\text{row}_1 = Z_{i_\text{value}} \times 1000 + (Z_{t_\text{value}} - Z_{i_\text{value}}) = 1 \times 1000 + (3 - 1) = 1002$$

$$\text{row}_3 = Z_{i_\text{value}} \times 1000 + (Z_{t_\text{value}} - Z_{i_\text{value}}) = 2 \times 1000 + (4 - 2) = 2002$$

$$\text{row}_4 = Z_{i_\text{value}} \times 1000 + (Z_{t_\text{value}} - Z_{i_\text{value}}) = 2 \times 1000 + (3 - 2) = 2001$$

Therefore, based on the `priority score`, $row_1$ is at the top of the candidate list, followed by $row_4$ and $row_3$. Now, we identify the `lczb` in $row_1$, which is found at the first column (if multiple `lczb` are found, we choose the leftmost one), and swapping the first and second columns increases the $Z_i$ for the row and $Z_p$ for the matrix from 1 to 2, consequently,

the swap is accepted by improve_Zp_permutation. We swap the columns and get a new column permutation [102345] and an updated matrix,

$$X = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}, [Z_t, Z_i] = \begin{bmatrix} 3,2 \\ 5,5 \\ 4,3 \\ 3,2 \end{bmatrix}$$

$row_1$ is added to the permanently_skipped array. We repeat the process of building the candidate list for the new permutation and iterating through the candidate list sequentially for $Z_p$ improvement. improve_Zp_permutation gets terminated once we go through the candidate list without obtaining any improvement and returns the final permutation of the matrix.

## 3.3 Exhaustive Approach

Among all the variants of GH, Exhaustive Approach searches both left and right sides of the lczb and consumes the most runtime. It starts with identifying the lczb for the selected row from the candidate list. Then, it picks the nearest two columns on the left side of the lczb, and swaps the columns to check whether it expands the length of the lczb (increasing $Z_i$) on the left side. It continues through the process if it gets an improved $Z_i$ for each swap without decreasing the $Z_p$ of the matrix. Once the left search gets terminated due to no improvement, it searches the right side of the lczb with a similar process, picks the nearest two columns on the right side of the lczb, and swaps the columns if $Z_i$ is improved without decreasing the current $Z_p$; otherwise, the Exhaustive approach terminates. Algorithm 3.6 explains the steps in the Exhaustive Approach.

## 3.4 Selective Exhaustive Approach

The `Selective Exhaustive` approach goes through the same process on the left side of the `lczb` for the selected row as the `Exhaustive` approach. However, it only searches through the right side of the `lczb` exhaustively for column swaps, only if zero improvement is found on the left side iteration. Algorithm 3.7 illustrates the process of the `Selective Exhaustive` Approach.

## 3.5 Single Attempt Approach

For the `Single Attempt` approach, the function (algorithm 3.8) identifies the `lczb` for the row, then picks the nearest two columns on the left side of the `lczb`, and only swaps the selected columns once. If $Z_p$ is improved, it terminates the search; otherwise, it goes through the same process on the right side of `lczb` and attempts a single swap. This variant of the `GH` is the most efficient in terms of the runtime of the program.

Algorithm 3.5: Improve-Zp-Permutation: Main algorithm for Zp optimization

```
IMPROVE-ZP-PERMUTATION(matrix, permutation, outfile, mode, quiet)
 1  rows = matrix.size
 2  Initialize permanently_skipped as empty set
 3  global_improved = TRUE, iteration = 0
 4  BestPermutation = permutation, BestZp = 0
 5  while global_improved
 6      iteration = iteration + 1
 7      global_improved = FALSE
 8      (Zmin, Zp, Zt_values, Zi_values) =
            EVALUATE-PERMUTATION(matrix, permutation)
 9      if Zp = Zmin // Optimal value of Z_p reached (Z_min)
10          Break
11      if Zp ≥ BestZp
12          BestZp = Zp, BestPermutation = permutation
13      candidates = GET-PRIORITIZED-CANDIDATES(Zi_values,
            Zt_values, Zmin, permanently_skipped, n_c, outfile)
14      for each row_idx in candidates
15          ok = FALSE
16          if mode = "exhaustive"
17              ok = TRY-IMPROVE-ROW-EXHAUSTIVE(matrix,
                permutation, row_idx, Zp, outfile)
18          elseif mode = "selective"
19              ok = TRY-IMPROVE-ROW-SELECTIVE-EXHAUSTIVE(matrix,
                permutation, row_idx, Zp, outfile)
20          else
21              ok = TRY-IMPROVE-ROW-SINGLE-ATTEMPT(matrix,
                permutation, row_idx, Zp, outfile)
22          if ok
23              global_improved = TRUE
24              Update the permutation
25              Break inner loop
26  return BestPermutation
```

Algorithm 3.6: Try-Improve-Row-Exhaustive: Multiple swap attempts for maximum improvement

```
TRY-IMPROVE-ROW-EXHAUSTIVE(matrix, perm, row_idx, Zp, outfile)
 1  row = matrix[row_idx], n = perm.size
 2  orig_perm = perm
 3  (Zmin, Zp, Zt, Zi) = EVALUATE-PERMUTATION(matrix, perm)
 4  init_Zi = Zi[row_idx]
 5  (start, len) = FIND-LONGEST-CONSECUTIVE-ZEROS(row, perm)
 6  if len = n or len = n − 1
 7      return FALSE
 8  L_imp = FALSE, R_imp = FALSE
 9  // Left expansion
10  L = (start − 1 + n) mod n, L_done = FALSE
11  while ¬L_done
12      c_L = perm[L]
13      if row[c_L] ≠ 1
14          L_done = TRUE; continue
15      swap_L = (L − 1 + n) mod n
16      if row[perm[swap_L]] = 0
17          new_perm = perm
18          swap(new_perm[L], new_perm[swap_L])
19          Evaluate new_perm and check acceptance
20          if accepted
21              Update perm, continue left
22          else L_done = TRUE
23      else L_done = TRUE
24  // Right expansion
25  R = (start + len) mod n, R_done = FALSE
26  (Zmin, Zp, Zt, Zi) = EVALUATE-PERMUTATION(matrix, perm)
27  while ¬R_done
28      c_R = perm[R]
29      if row[c_R] ≠ 1
30          R_done = TRUE; continue
31      swap_R = (R + 1) mod n
32      if row[perm[swap_R]] = 0
33          new_perm = perm
34          swap(new_perm[R], new_perm[swap_R])
35          Evaluate new_perm and check acceptance
36          if accepted
37              Update perm, continue right
38          else R_done = TRUE
39      else R_done = TRUE
40  return L_imp or R_imp
```

Algorithm 3.7: Try-Improve-Row-Selective-Exhaustive: Sequential left-then-right improvement

```
TRY-IMPROVE-ROW-SELECTIVE-EXHAUSTIVE(matrix, perm, row_idx, Zp, outfile)
 1  row = matrix[row_idx], n = perm.size
 2  orig_perm = perm
 3  (Zmin, Zp, Zt, Zi) = EVALUATE-PERMUTATION(matrix, perm)
 4  init_Zi = Zi[row_idx]
 5  (start, len) = FIND-LONGEST-CONSECUTIVE-ZEROS(row, perm)
 6  if len = n or len = n − 1
 7      return FALSE
 8  improved = FALSE
 9  // Left expansion
10  L = (start − 1 + n) mod n, left_done = FALSE
11  while ¬left_done
12      c_L = perm[L]
13      if row[c_L] ≠ 1
14          left_done = TRUE; continue
15      swap_L = (L − 1 + n) mod n
16      if row[perm[swap_L]] = 0
17          Attempt swap and check acceptance criteria
18          if accepted
19              Update perm, improved = TRUE
20          else left_done = TRUE
21      else left_done = TRUE
22  // Right expansion (if left failed)
23  if ¬improved
24      R = (start + len) mod n, right_done = FALSE
25      while ¬right_done
26          c_R = perm[R]
27          if row[c_R] ≠ 1
28              right_done = TRUE; continue
29          swap_R = (R + 1) mod n
30          if row[perm[swap_R]] = 0
31              Attempt swap and check acceptance criteria
32              if accepted
33                  Update perm, improved = TRUE
34              else right_done = TRUE
35          else right_done = TRUE
36  return improved
```

Algorithm 3.8: Try-Improve-Row-Single-Attempt: Single swap attempt for row improvement

TRY-IMPROVE-ROW-SINGLE-ATTEMPT($matrix, permutation, row_idx, current_Zp, outfile$)

1 $row = matrix[row_idx]$, $n = permutation.size$
2 $original_permutation = permutation$
3 $(original_Zmin, original_Zp, original_Zt, original_Zi) =$
    EVALUATE-PERMUTATION($matrix, permutation$)
4 $initial_Zi = original_Zi[row_idx]$
5 $(seg_start, seg_len) =$ FIND-LONGEST-CONSECUTIVE-ZEROS($row, permutation$)
6 **if** $seg_len = n$ or $seg_len = n - 1$
7     **return** FALSE
8 // Attempt left expansion once
9 $left_bound = (seg_start - 1 + n)$ mod $n$
10 $col_left = permutation[left_bound]$
11 **if** $row[col_left] = 1$
12     $swap_with = (left_bound - 1 + n)$ mod $n$
13     **if** $row[permutation[swap_with]] = 0$
14         $new_permutation = permutation$
15         swap($new_permutation[left_bound], new_permutation[swap_with]$)
16         $(new_Zmin, new_Zp, new_Zt, new_Zi) =$
            EVALUATE-PERMUTATION($matrix, new_permutation$)
17         **if** acceptance criteria satisfied
18             $permutation = new_permutation$
19             **return** TRUE
20 // Attempt right expansion once
21 $right_bound = (seg_start + seg_len)$ mod $n$
22 $col_right = permutation[right_bound]$
23 **if** $row[col_right] = 1$
24     $swap_with = (right_bound + 1)$ mod $n$
25     **if** $row[permutation[swap_with]] = 0$
26         $new_permutation = permutation$
27         swap($new_permutation[right_bound], new_permutation[swap_with]$)
28         $(new_Zmin, new_Zp, new_Zt, new_Zi) =$
            EVALUATE-PERMUTATION($matrix, new_permutation$)
29         **if** acceptance criteria satisfied
30             $permutation = new_permutation$
31             **return** TRUE
32 **return** FALSE

# Chapter 4

# Ant Colony Optimization

`Ant colony optimization (ACO)` is a meta-heuristic inspired by the foraging behavior of ants. To keep track of the path, real ants lay pheromone trails while traveling from the colony to the food source. Eventually, the pheromone concentration over the shortest path increases over time due to frequent use, leading other ants to follow the route; therefore, ants keep optimizing their path and shorten their tour distance. Marco Dorigo [6] first introduced `ACO` in the early 1990s and has been widely used in solving the Traveling Salesperson Problem (TSP), Job Scheduling Problem, Vehicle Routing Problem, etc. [5]. In our research, we utilize the functionality of a structured implementation of Marco et al. [7]. We frame the consecutive zeros problem (`CZP`) as a variant of TSP, and calculate distance according to the matrix. Further, we employ `ACO` on the matrix, which returns the optimized short path and provides a permutation of the columns for the matrix, which can be exploited to examine the effect of `ACO` in solving the consecutive zeros problem for the matrix.

## 4.1 TSP formulation & cost distance

The central idea of our heuristic methods is to represent the matrix column ordering problem as a variant of the Traveling Salesperson Problem (TSP). Considering each column as a unique city, we calculate the distance for each city to all other cities. If the number of columns/cities is $n_c$, the total number of distances

$$= \frac{n_c \cdot (n_c - 1)}{2}$$

We derive from the work of Chandra et al. [2], for every pair of columns $j,l \in n_c$ of matrix $B \in \mathbb{R}^{m \times n_c}$, we define distance,

$$c_{jl} = \sum_{i=1}^{m} B(i,j) \odot B(i,l), \tag{4.1}$$

where,

$$B(i,j) \odot B(i,l) = \begin{cases} 0, \text{ if } (B(i,j) = B(i,l) = 0) \text{ or } (B(i,j) \neq 0 \text{ and } B(i,l) \neq 0) \\ 1, \text{ otherwise} \end{cases} \tag{4.2}$$

Our objective is to find a permutation $\Pi$ of the columns employing ACO that minimizes the cost

$$C(B_\Pi) = \sum_{j=1}^{n_c - 1} c_{\Pi(j)\Pi(j+1)} \tag{4.3}$$

and maximizes `Z_p`.

The pairwise distance $c_{jl}$ is symmetric and nonnegative. Therefore, we can reduce the required storage from $O(n_c^2)$ to $O(n_c(n_c-1)/2)$ by saving the distances into a 1-dimensional array using the `Cost_array_index` (Equation 4.4). This preserves $O(1)$ access during construction and updates.

## 4.2 Cost Array Calculation

To cast the problem as a variant of TSP, for every pair of columns of a matrix, $B \in \mathbb{R}^{m \times n_c}$, the "distance or cost" is computed. Following obtaining the distance for all the columns, we store it using a 1-dimensional array taking the size of $= \frac{n_c \cdot (n_c - 1)}{2}$ instead of a 2-dimensional array of $n_c \cdot n_c$ sized matrix, where $n_c$ is the number of columns. The one-dimensional array exploits symmetry, thus reducing space complexity. To calculate the allocated index for the distance from city $i$ to $j$ or vice versa, we define,

$$\text{Cost_array_index}(i,j) = n_c \cdot i - \frac{i \cdot (i+1)}{2} + j - (i+1) \quad \text{here } i < j \tag{4.4}$$

Utilizing the cost index formula, we can map a 2-dimensional array to a 1-dimensional array with precision using the column indices $i$ and $j$ at any given time. In the case of $i > j$, we swap the column numbers as the distance from $i$ to $j$ is the same as the distance from $j$ to $i$, and proceed to calculate `Cost_array_index`.

## 4.3 Matrix Input

Our implementation of the `ACO` code has been configured to accept a special `.mtx` type file as input, in addition to the `.tsp` type file, which can contain the 2-dimensional matrix as input. Storing the matrix is crucial as calculating the $Z_p$ of the matrix requires the structural knowledge of the matrix at any given time. Listing 4.1 displays a sample input file for the `ACO`. The `TYPE` and `DIMENSION` fields indicate type and size of the input file, the `COMMENT` field provides crucial information for the matrix, such as `original matrix size`, `compressed matrix size`, $Z_{min}$, $Z_{max}$, $Z_p$, $A_0$ and $M_0$. $A_0$ and $M_0$ denote the average and median length of the maximum consecutive zeros in rows for the whole matrix. The input matrix is arranged between the fields `MATRIX` and `EOF`, where `EOF` represents the end of file.

Listing 4.1: Custom `.mtx` example (EisE)

```
NAME: EisE
TYPE: MATRIX
COMMENT: original matrix size 7x6, compressed matrix size 7x6,
Zmin 3, Zmax 4, Zp 2, A0 2.57143 M0 3
DIMENSION: 7 6
MATRIX:
 1 0 1 0 0 0
 0 1 0 0 1 0
 0 0 0 1 0 1
 0 1 1 1 0 0
```

```
 0 0 1 0 1 1
 1 1 0 0 0 1
 1 0 0 1 1 0
EOF
```

## 4.4 ACO Algorithm Overview

Based on the pheromone deposit, evaporation, ant-type, and selection method of the next city, different variants of `ACO` implementation prevail, where the most prominent ones are, namely, `Ant System (AS)`, `Elitist Ant System (EAS)`, `Rank-based Ant System (RAS)`, `MAX-MIN Ant System (MMAS)`, `Best-Worst Ant System (BWAS)`, `Ant Colony System (ACS)`, etc. Brief descriptions for the variants are listed in table 4.1.

Table 4.1: ACO variants at a glance (we focus on MMAS)

| Variant | Brief description |
|---|---|
| AS | All ants deposit $1/L$ on their tours after evaporation. |
| EAS | AS + extra deposits on global-best (“elitist”) edges. |
| RAS | Only top-$r$ ranked ants deposit with decreasing weights. |
| BWAS | Extra evaporation on iteration-worst edges; diversity boost. |
| ACS | Greedy rule with $q_0$ and local trail update $\tau \leftarrow (1-\xi)\tau + \xi\tau_0$. |
| MMAS | Bounds $\tau \in [\tau_{min}, \tau_{max}]$, best-only deposit, restarts. |

In our implementation, we primarily focused on `MMAS` as it imposes upper and lower bounds on the pheromone values using $\tau_{max}$ and $\tau_{min}$ to help regulate the influence of the pheromone on the ant’s decision within a feasible range, promoting a healthy balance between exploration and exploitation. Moreover, pheromone trails are reinitialized periodically to avoid stagnation in local optima, ideal for large-scale problems such as sparse matrices in our implementation. In `MMAS`, only the best solution found in each try deposits pheromone on the paths taken, intensifying search efforts around the best solution so far.

In our implementation, `ACO` sets the default parameters, as illustrated in Table 6.1. The program chooses the `MMAS` variant by default. Following the setting of defaults, the user

Algorithm 4.1: ACO_Main: Overall workflow

```
ACO_MAIN()
1  read custom .mtx instance // Read & preprocess
2  precompute pairwise costs d_ij
3  build distance-based-nearest-neighbor lists (size = nn_ants)
4  set parameters (m, n, α, β, ρ, n_ants, nn_ants, ...) // Initialize
5  initialize pheromone trails τ_ij for all (i, j)
6  for iter = 1 to max_iters
7       if early_stop is met
8           Break
9       clear tours and visited sets; choose start cities (random) // Reset ants
10      // Construct tour for each ant
11      CONSTRUCTSOLUTION(k)
12      // Evaluate
13      compute tour cost for each ant; set iteration-best; update best-so-far if improved
14      τ_ij = (1 − ρ) · τ_ij for all (i, j) // Pheromone evaporation
15      add Δτ on edges of selected reinforcing tour // Pheromone deposit
16 return best-so-far tour (column permutation), its cost, and recorded Zp
```

command is parsed to modify the existing parameters before proceeding with input processing. Overall workflow of ACO is illustrated in algorithm 4.1.

### 4.4.1 Algorithm for input parsing

ACO reads the custom .mtx file and stores the matrix as illustrated in algorithm 4.2. It stores the number of rows and columns into instance.matrix_rows and instance.matrix_cols respectively, which further is utilized to create a rows $\times$ columns sized matrix, instance.matrix to store the matrix. Then the value of entries for each row and column is iteratively parsed and imputed into the instance.matrix.

### 4.4.2 Algorithm for Distance Array

Following the input processing, ACO computes and stores the pairwise distance cost $d_{i,j}$ for any column $i, j \in n_c$ of a matrix $B \in \mathbb{R}^{m \times n_c}$ using the function COMPUTE_DISTANCES as illustrated in Algorithm 4.3 and the helper function MATRIX_DISTANCE, Algorithm 4.4.

Algorithm 4.2: ReadMatrix: Parse custom `.mtx` input

```
READMATRIX(matrix_file_name)
 1  f = fopen(matrix_file_name, "r")
 2  if f == NIL
 3      error "cannot open input file"; return error
 4  rows = 0; cols = 0; reading_matrix = FALSE
 5  // Scan header until literal MATRIX:
 6  while getline
 7      if line is blank
 8          continue
 9      if starts with NAME:
10      elseif starts with TYPE:
11      elseif starts with DIMENSION:
12          parse (rows, cols)
13          if parse fails or nonpositive
14              error "bad DIMENSION"; return error
15          instance.matrix_rows = rows
16          instance.matrix_cols = cols
17          instance.n = cols
18          instance.matrix = new rows × cols array of int, init to −1
19      elseif starts with EDGE_WEIGHT_TYPE:
20      elseif equals MATRIX:
21          reading_matrix = TRUE
22  if not reading_matrix
23      error "missing MATRIX:"; return error
24  // Read matrix body: exactly rows lines with cols integers each
25  for r = 0 to rows − 1
26      until nonblank
27      c = 0
28      while tokens remain and c < cols
29          parse integer val
30          if conversion fails
31              error "bad integer"; return error
32          instance.matrix[r,c] = val
33          c = c + 1
34      if c ≠ cols
35          error "incomplete row"; return error
36  fclose(f) // free any temporary buffer if needed
```

The COMPUTE_DISTANCES function creates a 1-dimensional array of size $= \frac{n_c \cdot (n_c - 1)}{2}$, and for each column pair of the matrix, calculates the distance as defined in section 4.1 using the function MATRIX_DISTANCE. The cost index equation 4.4 provides the index for the corresponding columns, which is used to store the distance for the columns in the cost_array.

Following the distance calculation and storage, all the ants, as per n_ants, are allocated memory with arrays of integer type, tour, character type, visited to keep track of their tour path and visited cities, and variable tour_length is used to store the tour length for all the ants. Additionally, two ants are created, namely best_so_far_ant and restart_best_ant, and memory is allocated respectively to update the best tour path and corresponding visited cities. In our implementation, the default value for n_ants is 25.

Algorithm 4.3: ComputeDistances: symmetric inter-city distance array

```
COMPUTEDISTANCES()
    Input: global instance.matrix with matrix_rows, matrix_cols (= n_c)
    Output: pointer D where D[l] stores distance for a column pair (i,j)
1   n_c = instance.matrix_cols
2   allocate D as a 1D array of length n_c(n_c − 1)/2 (type: long int)
3   for i = 0 to n_c − 1
4       for j = i to n_c − 1
5           index = Cost_array_index(i,j)
6           if i = j
7               D[index] = 0
8           else
9               d = MATRIXDISTANCE(i,j)
10              D[index] = d
11  return D
```

### 4.4.3 Nearest-neighbor candidate lists

For speed and exploitation, ACO builds Nearest-neighbor candidate lists for all the cities to limit the exhaustive search during solution construction. Based on distance in-

Algorithm 4.4: MatrixDistance($i$, $j$): Row-wise difference count between columns $i$ and $j$

```
MATRIXDISTANCE(i,j)
    Input: column indices i, j; global instance.matrix with matrix_rows, matrix_cols
    Output: integer distance = number of rows where entries differ
1   distance = 0
2   if i ≥ instance.matrix_cols or j ≥ instance.matrix_cols
3       error "index out of bounds"; return error
4   for row = 0 to instance.matrix_rows − 1
5       if instance.matrix[row] == NULL
6           error "null row pointer"; return error
7       if instance.matrix[row,i] ≠ instance.matrix[row,j]
8           distance = distance + 1
9   return distance
```

formation, ACO creates Nearest-neighbor candidate list, $N_i$ for city $i$, where it picks the top nn_ants closest cities by distance from $i$ ($i \in n_c$ and $n_c$ is the number of cities). In our implementation, the size of $N_i$ is the same as nn_ants, which is set to 20 if the $n_c \geq 20$ of the matrix, otherwise nn_ants is set to $n_c - 1$. During solution construction, we iterate through $N_i$; if $N_i$ is exhausted, we fall back to all remaining cities. Algorithm 4.5 illustrates the process of building Nearest-neighbor candidate lists.

Algorithm 4.5: Build Nearest Neighbor Lists

```
BUILDNNLISTS(n_c, nn_ants)
1   for i = 0 to n_c − 1
2       x = collect pairs (j, d_ij) for j ≠ i
3       x = sort x by d_ij ascending
4       N_i = first min(nn_ants, n_c-1) indices of x
```

### 4.4.4 Pheromone and distance heuristic initialization

Following generating `Nearest-neighbor candidate lists`, `ACO` iterates `max_tries` times for solution construction, and for each trial, `ACO` first initializes the pheromone trails for all the edges according to the variant. Our `ACO` utilizes `Min-Max Ant System (MMAS)`, which initializes the pheromone trails using $\tau_{max}$,

$$\tau_{\text{max}} = \frac{1}{\rho \cdot \text{nn_tour()}} \qquad \tau_{\text{min}} = \frac{\tau_{\text{max}}}{2 \cdot n_c} \tag{4.5}$$

Where:

- $\tau_{\text{max}}$ and $\tau_{\text{min}}$ are the maximum and minimum pheromone value for `MMAS`. $\rho$ is the pheromone evaporation rate (0.5 for `MMAS`).
- $n_c$ is the number of columns/cities, $nn_tour()$ is the distance of a greedy tour, constructed by choosing the closest columns by distance.

For all the column $i, j \in n_c, i \neq j$, of a matrix, $B \in \mathbb{R}^{m \times n_c}$, `ACO` calculates `Total_Information`, a combined score consists of pheromone concentration $\tau$ and distance heuristics $\eta$, factoring with $\alpha$ and $\beta$ respectively. `Total_Information` is the driving factor while selecting the next city, indicating the eligibility of the candidate city to be chosen.

$$\text{Total_Information}_{ij} = (\tau_{ij})^{\alpha}(\eta_{ij})^{\beta} \tag{4.6}$$

$$\text{Distance Heuristic, } \eta_{ij} = \frac{1}{d_{ij} + \varepsilon} \tag{4.7}$$

$\tau_{ij}$ and $\eta_{ij}$ are the values for pheromone concentration and distance heuristics for city $i$ to $j$ and vice versa. $d_{i,j}$ is the distance between column $i$ and $j$, and $\varepsilon$ is a small constant ($10^{-9}$) to avoid division by zero in $\eta_{ij}$.

### 4.4.5 Solution construction (roulette on $S_i$)

For each trial up to `max_tries`, until the termination condition is met, `ACO` loops through and continues to build `construct_solutions`, employs `local_search`, updates `statistics` and `pheromone_trails`. Each iteration of the loop, `construct_solutions` function (algorithm 4.6) is utilized to generate complete tours for all the ants; therefore, we obtain `n_ants` number of solution paths in each iteration. To construct a solution, `ACO` starts by erasing the memory of all ants. Using the `Seed`, it generates a random number between $[0, n_c - 1]$ ($n_c$ is the number of cities) and places all the ants on that same randomly chosen city to start with. Then, until the tour is complete, it calls the `neighbor_choose_and_move_to_next` function (function procedure illustrated in algorithm 4.6) for all the ants to pick the next city from the candidate list of cities. Therefore, all the ants build their path in parallel.

For selecting the next city from the current city $i$, `ACO` first considers the `Nearest-neighbor candidate list`, $N_i$ for city $i, i \in n_c$. In the function `neighbor_choose_and_move_to_next`, the `cumulative_sum`, $S_i$, is calculated by adding the `Total_Information(i, candidate_city)` (equation 4.6) for all the unvisited candidate cities in $N_i$. Given that the $S_i > 0.0$, `ACO` draws a random number, $rnd \in [0, S_i]$. Now, `ACO` walks through the candidates that formed $S_i$ in index order and sequentially adds the `Total_Information` of the candidate cities to `partial_sum`, $p_i$. The roulette continues until $p_i \leq rnd$. Once adding `Total_Information` of a candidate city to $p_i$ fulfills the condition $p_i > rnd$, the roulette breaks and the associated candidate city is selected as the next city. This roulette on choosing the next city helps `ACO` to explore new paths effectively, rather than relying solely on pure greedy heuristics, which can potentially lead to better path construction.

In the case of `ACO` being exhausted of the neighbor list, $N_i$, it calls the `choose_best_next` function, which acts based on pure pheromone and distance heuristic information and selects the candidate with higher `Total_Information` (equation 4.6) out of all the cities.

Algorithm 4.6: Construct-Solutions: Build tour for ants

```
CONSTRUCTSOLUTIONS()
1  choose start city randomly, start; set k = start // Initialize
2  tour = [start];   visited[start] = TRUE
3  for each ants:
4      i = k
5      while |tour| < n
6          cand = N_i ∩ {unvisited} // Candidate set from unvisited nearest neighbors
7          if cand = ∅
8              cand = {all unvisited cities}
9          sum = 0; Partial_Sum = = 0
10         for each j ∈ cand
11             s_ij = τ_ij^α η_ij^β; sum = sum + s_ij // Compute total information s_ij
12         // Selecting next city
13         Rnd ∈ [0, sum]
14         for each j ∈ cand
15             if Partial_Sum > Rnd
16                 j = city fulfilling the breaking condition; Break;
17             Partial_Sum = Partial_Sum + s_ij
18         append j to tour;   visited[j] = TRUE
19         i = j
20 if close_tour // Close tour if required; evaluate
21     append start to tour
22 compute cost = tour cost; record any Zp metrics
23 return (tour, cost)
```

### 4.4.6 Local search

After constructing each tour, a lightweight local search (2-opt, 2.5-opt, or 3-opt) may be applied depending on the user command, which attempts to swap cities within the solution obtained to determine whether the path can be further improved to reduce the cost, $C(A_{\Pi})$. For the 2-opt local search, two cities are removed from the path, their positions swapped, and they are reinserted into the solution path. In contrast, for the 2.5-opt, a city is removed from the path and inserted between the other two consecutive cities. 3-opt local search identifies 3 distinct connections and cuts the path into 3 segments, then swaps the segments. Swap is accepted, given that it reduces the cost of the path. Local search

is limited to only checking for improvement within the `Nearest-neighbor candidate list` after obtaining the first random city. If any city does not yield any improvement, Local search skips the city in subsequent iterations, and marks the city `TRUE` using a boolean array size of cities, `DLB`. Local search terminates if a full randomized pass over all cities (skipping cities with `DLB=TRUE`) yields no improvement. In our runs, we enable 3-opt local search.

### 4.4.7 MMAS pheromone model

Following the `construct_solutions` and `local_search`, `ACO` updates the `statistics` and `pheromone_trails`. For `statistics`, `best_so_far_ant`, and `restart_best_ant`, ants are updated if a better tour is found with a shorter path distance in the iteration. Moreover, as `MMAS` constrains trails to $[\tau_{\min}, \tau_{\max}]$, `ACO` re-initializes $\tau_{\max}$ and $\tau_{min}$ with the tour length of `best_so_far_ant`, reinforcing only the best tour. Let $L_{\text{best}}$ be the cost of the tour for `best_so_far_ant` and $\rho$ is the evaporation rate. We calculate $\tau_{\min}$ and $\tau_{\max}$,

$$\tau_{\max} = \frac{1}{\rho L_{\text{best}}}, \qquad \tau_{\min} = \frac{\tau_{\max}}{2n}, \tag{4.8}$$

For `pheromone_trail` updates, Global pheromone evaporation in our implementation:

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} \tag{4.9}$$

here, $\tau_{ij}$ is the pheromone concentration between city i and j, $i, j \in n_c, i \neq j$, $n_c$ is the number of cities. As we are using `MMAS`, in case of $\tau_{ij} < \tau_{\min}$ after the evaporation, $\tau_{ij}$ is updated with $\tau_{\min}$. Then, the `pheromone_trails` is updated only with the tour length of the best ant so far. Let,

$$d_\tau = 1.0/L_{\text{best}} \tag{4.10}$$

We use $d_\tau$ to update the pheromone trails of the best tour ant.

$$\tau_{ij} \leftarrow \tau_{ij} + d_\tau \tag{4.11}$$

Following the update of the `pheromone_trails`, `Total_Information` is recalculated, reflecting the effect of updated `pheromone_trails` for future solution constructions.

### 4.4.8 Termination Criteria

In our implementation, the trial goes through the specified number of *max_tries*. For each trial of the run, the run terminates on reaching `max_tours` and `max_time`, or `tour_length` has reached optimal. Until then, the iteration continues and each iteration goes through the process of building `construct_solutions`, employing `local_search`, updating `statistics` and `pheromone_trails`. We obtain `n_ants` (number of ants) new solution paths in each additional iteration.

### 4.4.9 Complexity and memory

The `distance_array`, `pheromone_trails` and `Total_Information`, costs $O(n_c^2)$ memory for the matrix $B \in \mathbb{R}^{m \times n_c}$. Each iteration performs $O(n_c \cdot \texttt{nn_ants})$ scoring per ant during construction (or $O(n_c^2)$ if falling back to all cities), plus $O(n_c \cdot nn_ants)$ for evaporation and $O(n_c)$ for deposit of pheromone on the best tour. However, for the aforementioned square matrices, we use the triangular storage (1-dimensional array) that requires $n_c(n_c - 1)/2$ slots instead of an $n_C \times n_c$ matrix for $n_c$ columns/cities. Thus, we reduce the memory and time complexity during the execution of the program.

# Chapter 5

# Hybrid ACO and Zp Estimator

In this chapter, we introduce two $\mathtt{Z_p}$-aware variants of the ACO, derived from the implementation of ACO described in the previous chapter. As we aim to obtain $B_\pi$, with a column permutation $\pi$ for a matrix $B \in \mathbb{R}^{m \times n_c}$, such that $\mathtt{Z_p}$ of $B_\pi$ is maximized, therefore, apart from just pheromone and pure distance heuristic-based approaches, we formulated metaheuristics $\mathtt{Z_p}$-Hybrid and $\mathtt{Z_p}$-Estimator that are guided by $\mathtt{Z_p}$ score during solution construction.

## 5.1 Zp-Hybrid

Like the ACO metaheuristic described in Chapter 4, our $\mathtt{Z_p}$-Hybrid implementation also accepts a special type of .mtx file, as defined in Section 4.3 and illustrated in Listing 4.1, alongside .tsp files. We use MMAS as the default method for this variant of the ACO. The solution is driven by the $\mathtt{Z_p}$ score; therefore, we do not use any local search after every solution construction, as it alters the $\mathtt{Z_p}$ improvement if found. During solution construction, we track and save the best $\mathtt{Z_p}$ score using the integer variable best_zp_value, and the corresponding ant is stored in best_zp_ant.

### 5.1.1 Calculating Zpair

To bias our solution towards $\mathtt{Z_p}$ improvement, for all the column pairs in the matrix, $\mathtt{Z_p}$-Hybrid calculates Zpair and stores in $\mathtt{Z_p}$-array, an integer array of size $\frac{n_c \cdot (n_c - 1)}{2}$ given that $n_c$ is the number of columns, where the index of the array is determined using the

`Cost_array_index`, equation 4.4 from chapter 4. `Zpair` value between the two columns $i$ and $j$ can be represented as

$$ZPair_{ij} = \sum_{k=1}^{m} \mathbf{1}(x_{ki} = 0 \wedge x_{kj} = 0) \tag{5.1}$$

where,

- m is the total number of rows.
- k is the row index; $x_{ki}$ and $x_{kj}$ are the values in column $i$ and $j$ for row $k$.
- here **1** represents a function, which returns 1 if the $x_{ki} = 0 \wedge x_{kj} = 0$ is true and 0 otherwise.
- $\wedge$ represents the logical AND operator.

Following is an illustrative example for *Zpair_score* calculation for *ColumnA* and *ColumnB*.

| COLUMN A | COLUMN B | Zpair_Count |
|---|---|---|
| 0 | 0 | +1 |
| 0 | 1 | +0 |
| 1 | 1 | +0 |
| 0 | 0 | +1 |
| 1 | 1 | +0 |
| 0 | 0 | +1 |
| 0 | 0 | +1 |
| 0 | 0 | +1 |
| 1 | 1 | +0 |
| 0 | 0 | +1 |

$$\text{Zpair_Score} = \frac{\text{Zpair_Count}}{\text{Row}} = \frac{6}{10} = 0.6$$

Algorithm 5.1: ComputeZpair: Upper-triangular co-zero counts

```
COMPUTEZPAIR()
1   R = instance.matrix_rows;    n_c = instance.matrix_cols
2   allocate Zp-array as array of length n_c(n_c - 1)/2
3   for i = 0 to n_c - 2
4       for j = i + 1 to n_c - 1
5           Zero_Pair_Count = 0
6           for r = 0 to R - 1
7               if instance.matrix[r,i] == 0 ∧ instance.matrix[r,j] == 0
8                   Zero_Pair_Count = Zero_Pair_Count + 1
9           l = Cost_array_index(i, j) // same index as cost array
10          Zp-array[l] = Zero_Pair_Count / R // normalized co-zero frequency
11  return Zp-array
```

Prior to storing in the $Z_p$-array, the Zpair value is normalized using the number of rows of the matrix. The $Z_p$-array is further utilized to create the Zpair-based-neighbor-list, $Z_p$_nn, which contributes in creating Hybrid-neighbor-list, We use Hybrid-neighbor-list during solution construction to guide the solution towards $Z_p$ improvement. Algorithm 5.1 illustrates the process of populating the $Z_p$-array.

### 5.1.2 Zpair-based nearest-neighbor lists

Once the Zpair is calculated, $Z_p$-Hybrid creates a nearest-neighbor list, $Z_p$_nn for all the columns based on the Zpair values. For any given column $i$ of a matrix $i \in n_c, B \in \mathbb{R}^{m \times n_c}$, it selects the $k$ number of columns from the set of all other columns, $\{j \mid 0 \leq j \leq n_c - 1, j \neq i\}$, that have the highest Zpair value for $i$ and stores in the $Z_p$_nn(i); hence, the $Z_p$_nn consist of columns that has higher probability to form consecutive sequence of zeros if placed next to corresponding column during solution construction. Algorithm 5.2 displays the workflow of building Zpair-based-neighbor-lists; here, $k$ is set to 10.

$$\text{Zp_nn}(i) = \text{TopK}\Big\{ ZPair_{ij} \ : \ j \in \{0, \ldots, n-1\} \setminus \{i\} \Big\}. \tag{5.2}$$

Algorithm 5.2: BuildNNLists_Zpair($k$): Top-$k$ neighbors per column by *ZPair*

```
BUILDNNLISTS_ZPAIR(k)
1   C = instance.matrix_cols
2   allocate nn as an array of C pointers of length k
3   for i = 0 to C − 1
4       allocate nn[i] as an array of k long int
5       used = 0
6       while used < k
7           best = −1;   bestv = −1.0
8           for j = 0 to C − 1
9               if j == i
10              else
11                  // skip if j already selected for row i
12                  already = FALSE
13                  for t = 0 to used − 1
14                      if nn[i][t] == j
15                          already = TRUE
16                  if already: continue // do nothing
17                  else
18                      v = instance.zpair[i, j]
19                      if v > bestv
20                          best = j;   bestv = v
21          if best == −1
22              nn[i][used] = i // fallback/self if insufficient distinct neighbors
23          else
24              nn[i][used] = best;   used = used + 1
25  return nn
```

### 5.1.3 Hybrid Neighbor Lists

Apart from the $Z_p$_nn, $Z_p$-Hybrid also computes the distance-based-neighbor-list, similar to the ACO implementation in chapter 4. For each city, we combine both the neighbor-lists and create Hybrid-neighbor-list. In this work, the size of both the lists is 20, contributing towards the max size of 40 combined for the Hybrid-neighbor-list.

Let $i$ be the column we are building the Hybrid-neighbor-list for, which combines the $Z_p$_nn and the distance-based-neighbor-list list, selecting only the unique

columns from both lists. For the duplicate cities in both lists, the Hybrid-neighbor-list contains empty spots at the end of the array, which are padded with the current column $i$. The process is further explained in algorithm 5.3.

$$Hybrid_nnlist(i)^{\star} \;=\; \mathrm{Zp_nn}(i) \,\cap\, \mathrm{nn_list_distance}(i) \tag{5.3}$$

$$Hybrid_nnlist(i) \;=\; \mathrm{pad}_0^P\big(Hybrid_nnlist(i)^{\star}, i\big) \tag{5.4}$$

Here, p is the length difference of $Hybrid_nnlist(i)$ and $Hybrid_nnlist(i)^{\star}$

### 5.1.4 Zp-aware Solution Construction

$\mathrm{Z_p}$-Hybrid goes through the same process of constructing solutions and follows the same termination criteria as ACO, described in chapter 4. However, unlike ACO, the $\mathrm{Z_p}$-Hybrid iterates through the Hybrid-neighbor-list (subsection 5.1.3) during solution construction, instead of distance-based-neighbor-list. Moreover, to generate the solution more $\mathrm{Z_p}$ orientated, we introduce a new Total_Information formula, an upgrade over the formula 4.6 in chapter 4, for any two columns $i$ and $j$, the updated Total_Information,

$$\mathrm{Zp_Total_Information}_{ij} = (\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}(ZPair_{ij} + \varepsilon)^{\beta_{zp}} \tag{5.5}$$

Here, $\varepsilon$ is a very small positive number, set to $1 \times 10^{-9}$, $\beta_{zp}$ is the factor to influence the integration of $\mathrm{Z_p}$ in the solution, set to 1 in this implementation. We use the updated $\mathrm{Z_p}$_Total_Information during solution construction.

After every solution path obtained, $\mathrm{Z_p}$ for the tour is calculated as algorithm 5.4, for each tour construction, best_zp_value is tracked based on the best $\mathrm{Z_p}$ score, and the best_zp_ant and best_so_far_ant are updated accordingly.

Algorithm 5.3: BuildNNLists_Hybrid($k_{\text{dist}}, k_{\text{zp}}$): Merge Zpair-NN and distance lists

```
BUILDNNLISTS_HYBRID(k_dist, k_zp)
    Input: C = instance.n, instance.nn_list (distance NN), instance.nn_list_zp (Zpair NN)
    Output: hy[0..C-1][0..K-1], K = k_dist + k_zp
 1  C = instance.n;   K = k_dist + k_zp
 2  allocate hy as an array of C pointers to K long int
 3  for i = 0 to C − 1
 4      allocate hy[i] as an array of K long int
 5      u = 0
 6      for ii = 0 to k_zp − 1 // copy from Zpair-based NN, excluding self
 7          if u < K
 8              v = instance.nn_list_zp[i, ii]
 9              if v == i
10              else
11                  dup = FALSE
12                  for t = 0 to u − 1
13                      if hy[i][t] == v
14                          dup = TRUE
15                  if ¬dup
16                      hy[i][u] = v;  u = u + 1
17      for ii = 0 to min(k_dist, nn_ants) − 1 // copy from distance-based NN (cap by nn_ants)
18          if u < K
19              v = instance.nn_list[i, ii]
20              dup = FALSE
21              for t = 0 to u − 1
22                  if hy[i][t] == v
23                      dup = TRUE
24              if ¬dup
25                  hy[i][u] = v;  u = u + 1
26      while u < K // pad with self if still short (rare)
27          hy[i][u] = i;  u = u + 1
28  return hy
```

Algorithm 5.4: ComputeZpForTour: Wrap-aware $Z_p$ for a finished permutation

```
COMPUTEZPFORTOUR(tour)
    Input: tour is a column permutation array, matrix with R rows and C columns
    Output: Zp is the minimum over all rows of the maximum run of zeros
1   R = instance.matrix_rows;   C = instance.matrix_cols
2   Zp = C // upper bound for Z_p
3   for r = 0 to R − 1
4       lead = 0;   trail = 0;   run = 0;   maxrun = 0
5       for j = 0 to C − 1 // count leading zeros
6           col = tour[j];   bit = instance.matrix[r, col]
7           else Break
8       for j = C − 1 downto 0 // count trailing zeros
9           col = tour[j];   bit = instance.matrix[r, col]
10          else Break
11      for j = 0 to C − 1 // count maximum internal zero run
12          col = tour[j];   bit = instance.matrix[r, col]
13          if bit == 0
14              run = run + 1
15          else
16              run = 0
17      Zi = max(lead + trail, maxrun)
18      if Zi < Zp
19          Zp = Zi
20  return Zp
```

### 5.1.5 Zp-aware Pheromone Update

Following the solution construction for ants, $\mathrm{Z_p}$-Hybrid evaporates the pheromone trails similar to ACO in chapter 4. However, to update the pheromone trail with respect to $\mathrm{Z_p}$, we define,

$$d_\tau = \frac{\text{best_zp_value} + 1.0}{n_c + 1.0} \tag{5.6}$$

Using the $d_\tau$, we deposit extra pheromone on best_zp_ant's path as algorithm 5.5, incentivizing the route where we got the best $\mathrm{Z_p}$ so far; therefore, the future tours can get biased towards the path. $d_\tau$ is normalized by adding 1 in the numerator and denominator. Following the pheromone update, Zp_total_information is recalculated, and the iteration

goes on until termination for the specified number of trials.

Algorithm 5.5: Global Update Pheromone Add: Reinforce pheromone based on tour

```
GLOBALUPDATEPHEROMONEADD(a, d_tau)
    Input: ant a, delta pheromone d_tau
    Output: updates the pheromone trail of each edge in the tour
1   for i = 0 to n − 1
2       j = a.tour[i]
3       h = a.tour[i + 1]
4       l = Cost_array_index(i, h) // same index as cost array
5       pheromone[l] = pheromone[l] + d_tau
6   return
```

## 5.2 Zp-Estimator

Same as the ACO, the special .mtx type (section 4.3) input is accepted and processed in $Z_p$-Estimator. Moreover, the distance-based-neighbor-list and MMAS variant of the ACO are utilized. For solution construction and termination condition, we follow the same process as ACO for the $Z_p$-Estimator. An integer variable best_zp_value and a new ant, best_zp_ant, are introduced to keep track of the best $Z_p$ value and the corresponding ant during solution construction. Additionally, for this robust implementation, a few more parameters for each ant are initiated as described in table 5.1. No local search was used in this variant of the implementation, maintaining the focus on $Z_p$-aware solutions.

Table 5.1: Zp–specific fields stored per ant (ant_struct)

| **Field** | **Type/shape** | **Updated by** | **Meaning / role** |
|---|---|---|---|
| lead[r] | int, length $R$ | init, append | Length of the *leading* zero run at the tour start. Grows until '1' has been seen (lead_open=1). |
| trail[r] | int, length $R$ | init, append | Length of the *trailing* zero run at the tour end. Resets to 0 on appending a 1. |
| maxrun[r] | int, length $R$ | init, append | Maximum *internal* zero run observed so far in the prefix. |
| lead_open[r] | bool, length $R$ | init, append | Flag: 1 if the leading run is still contiguous (no 1 seen yet in row $r$) otherwise 0. Controls whether lead can grow. |
| seen_one[r] | bool, length $R$ | init, append | Flag: 1 if a nonzero has been seen (prevents double–counting lead+trail before wrap). |
| zp_now | long int | init, append | Current $Z_p$ without column wrap-around: $\min_r$ maxrun$[r]$ no wrap until final step. |
| zp_final | long int | finalize | Final wrap–aware $Z_p$ after tour completes. |

### 5.2.1 First city initialization and parameter update

After the initial setup of data structures, for each trial, $Z_p$-Estimator constructs solutions, updates pheromone_trails, and tracks the best_zp_value and the best_zp_ant until the termination condition is met. Before solution construction, $Z_p$-Estimator initializes the pheromone_trails, computes Total_Information for the columns. During solution

construction, all the ants are placed on a randomly selected city/column, $p, p \in n_c$, for the matrix $A \in \mathbb{R}^{m \times n_c}$, and subsequently, the function `dzp_int_after_first_city` is executed for all the ants. For each row in the selected column $p$, `dzp_int_after_first_city` function initializes `lead`, `trail`, `maxrun`, and `lead_open` to 1, and `seen_one` to 0 for the row, if the value at the row position is 0 for $p$, otherwise, all the parameters are set to 0 except `seen_one` is set to 1. Then, for all the ant, `Zp_now` is updated with the minimum of all the row `maxrun` (current `Zp`) values. With every solution step, all the ant parameters keep track of the current state of the solution. A detailed illustration of the parameter initialization process is provided in the algorithm 5.6.

Algorithm 5.6: Dzp_init_after_first_city: initialize per–row Zp state

```
DZP_INIT_AFTER_FIRST_CITY(a, first_city)
     Input: ant a; first column first_city; matrix A of size R × C
     Output: initialize a's rowwise counters and a.zp_now
 1   R = instance.matrix_rows;    C = instance.matrix_cols
 2   minmax = +∞
 3   for r = 0 to R − 1
 4       if instance.matrix[r, first_city] == 0
 5           a.lead[r] = 1;    a.trail[r] = 1;    a.maxrun[r] = 1
 6           a.lead_open[r] = 1;    a.seen_one[r] = 0
 7       else
 8           a.lead[r] = 0;    a.trail[r] = 0;    a.maxrun[r] = 0
 9           a.lead_open[r] = 0;    a.seen_one[r] = 1
10       if a.maxrun[r] < minmax
11           minmax = a.maxrun[r]
12   a.zp_now = minmax
13   return
```

### 5.2.2 Zp-Estimator Solution Construction

During solution construction, for each step of path building, the `Zp-Estimator` calls the `neighbor_choose_and_move_to_next` function to obtain the next eligible city, similar to `ACO` subsection 4.4.5. However, the function is tailored in `Zp-Estimator` implementation to estimate the improvement of `Zp` upon adding a candidate column to the current path. The function iterates all the unvisited columns from the `distance-based-neighbor-list`

and projects the updated value of `Zp` for the selected column if the column is appended. To fetch the projected `Zp`, `Zp`$^{\texttt{proj}}$, the `dzp_projected_Zp_if_append` (algorithm 5.7) function is used. Using the function, we can measure the effect, $\delta_zp$, of the candidate city if appended, we denote,

$$\delta_zp \;=\; \exp\!\left(\gamma \times \left(Z_p^{\text{proj}} - Z_p^{\text{now}}\right)\right) \tag{5.7}$$

Here, $\gamma$ is the factor to influence the Zp-aware selection of the next column; a higher value produces more exploitation.

Algorithm 5.7: Dzp_projected_Zp_if_append: look-ahead $Z_p$ if appending *candidate*

```
DZP_PROJECTED_ZP_IF_APPEND(a, cand, phase)
     Input: ant a; candidate column cand; construction phase phase
     Output: projected wrap-aware Zp if cand is appended at phase
 1   R = instance.matrix_rows;   C = instance.matrix_cols
 2   is_last = (phase == C − 1)
 3   minproj = +∞
 4   for r = 0 to R − 1
 5       if instance.matrix[r, cand] == 0
 6           trial_trail = a.trail[r] + 1
 7       else
 8           trial_trail = 0
 9       trial_max = max(a.maxrun[r], trial_trail)
10       wrapped = trial_max
11       if is_last
12           combine = trial_trail + a.lead[r]
13           row_all_zero = (a.seen_one[r] == 0) ∧ (instance.matrix[r, cand] == 0)
14           if row_all_zero
15               wrapped = C // entire row zero after closing
16           else
17               if combine > wrapped
18                   wrapped = combine
19       if wrapped < minproj
20           minproj = wrapped
21   return minproj
```

Using the $\delta_zp$, the probability is stored for all the unvisited candidates, then the probabilities are used to determine the next city using the same roulette process shown in sub-

section 4.4.5 in chapter 4. In the case of the `distance-based-neighbor-list` getting exhausted, the `Zp-Estimator` falls back to iterate through all unvisited columns in the matrix. Upon selecting a candidate $p$ as the next column, the function `dzp_update_after_append` updates the ant parameters for all the rows. For the appended column $p$, if the value at column $p$ and row position $k$ is 0, then `trail[k]` is increased by 1; if `lead_open[k]` is still 1, then `lead[k]` is also increased by 1, otherwise, `trail[k]` is set to 0, `lead_open[k]` to 0 and `seen_one` to 1. Moreover, the `dzp_update_after_append` function updates `max_run` parameter by `trail[k]` length if `trail[k] > max_run`. Following the parameter updates for all the rows, `Zp_now` is updated using `max_run` of the rows to reflect the overall `Zp` of the current constructed matrix. A detailed analysis of the function is provided in algorithm 5.8.

Algorithm 5.8: Dzp_update_after_append

```
DZP_UPDATE_AFTER_APPEND(a, appended_city)
    Input: ant a; newly appended column appended_city
    Output: updates a.lead, a.trail, a.maxrun, a.zp_now
 1  R = instance.matrix_rows
 2  minmax = +∞
 3  for r = 0 to R − 1
 4      if instance.matrix[r, appended_city] == 0
 5          a.trail[r] = a.trail[r] + 1
 6          if a.lead_open[r] == 1
 7              a.lead[r] = a.lead[r] + 1
 8      else
 9          a.trail[r] = 0
10          a.lead_open[r] = 0
11          a.seen_one[r] = 1
12      if a.trail[r] > a.maxrun[r]
13          a.maxrun[r] = a.trail[r]
14      if a.maxrun[r] < minmax
15          minmax = a.maxrun[r]
16  a.zp_now = minmax
17  return
```

### 5.2.3 Zp Finalize tour

As we consider the $Z_p$ with the column wrap around, following the completion of tour construction, we calculate the final $Z_p$ of the result matrix utilizing the function `dzp_finalize_tour`. The function updates the wrap-around final $Z_p$ of the result matrix to the `zp_final` variable for all the ants. A detailed analysis of the function is provided in algorithm 5.9. Following the solution construction and finalizing the `zp_final` for all the ant tours, `best_zp_value` and `best_zp_ant` are updated from the best ant tour so far using the function `find_best_by_zp` (algorithm 5.10).

Algorithm 5.9: Dzp_finalize_tour: compute and store final wrap–aware $Z_p$ for the finished tour

```
DZP_FINALIZE_TOUR(a)
     Input: ant a with per–row state after full tour construction
     Output: sets a.zp_final to min_r max(maxrun[r], lead[r] + trail[r])
 1   if ¬instance.matrix
 2       a.zp_final = 0
 3       return
 4   R = instance.matrix_rows
 5   minv = +∞
 6   for r = 0 to R − 1
 7       wrap = a.maxrun[r]
 8       combo = a.lead[r] + a.trail[r]
 9       if combo > wrap
10           wrap = combo
11       if wrap < minv
12           minv = wrap
13   a.zp_final = minv
14   return
```

Algorithm 5.10: FindBestByZp: return index of ant with maximal final $Z_p$

```
FINDBESTBYZP()
    Input: array ant[0..n_ants-1] with field zp_final
    Output: index k_best maximizing ant[k].zp_final
1   k_best = 0
2   best = ant[0].zp_final // maximize Z_p
3   for k = 1 to n_ants − 1
4       if ant[k].zp_final > best
5           best = ant[k].zp_final;    k_best = k
6   return k_best
```

# Chapter 6

# Experimental Evaluation

All the test matrices in our research were collected in matrix market format from the renowned public repository, SuiteSparse Matrix Collection [16]. Our study leveraged a dataset comprising 111 distinct sparse matrices from the repository. The matrices were sourced from real-world applications across a range of scientific domains, including, but not limited to, Linear Programming, Structural Engineering, Combinatorial Problems, and Chemical Process Simulation. To check the eligibility of the test cases for our problem, we first compress the matrix using the state-of-the-art coloring algorithms from `DSJM` [9]. Then we accept the matrix for evaluation if the compressed matrix has $Z_{min} > 1$. Matrices having $Z_{min} == 1/0$ give us no room for $Z_p$ improvement with column permutation; therefore, we exclude them from our test cases. We observe that 50 out of 111 matrices are eligible for our research (_T suffix in matrix name denotes transpose of a matrix). Further, we augment our test set with our `Constructed Matrices`, presented in table 6.2. All the experiments were performed using a laptop with a 3.80 GHz Intel(R) Core(TM) Ultra 7 155H CPU, 8 GB RAM, 1.23 MB L1 cache (528 KB L1d + 704 KB L1i), 22 MB L2 cache, and 24 MB L3 cache running Linux.

In this Chapter, we categorize our outputs into four sections. Firstly, we analyze the performance of `Greedy Heuristics`, followed by discussing the effect of `ACO` on `Constructed Matrices`. Then, we compare our results with outputs of Nearest Neighbor Heuristic and 2-opt algorithm from Chandra et al. [2], and finally conclude the chapter by examining the improvement of $Z_p$ using our proposed models on the remaining test metrics. The parame-

ters required to represent the results are:

- $d_0$: $Z_p$ of the natural ordering of the matrix following the compression scheme,
- $d_1$: $Z_p$ of the Nearest-neighbor heuristic ([2]),
- $d_2$: $Z_p$ of the 2-opt algorithm ([2]),
- $d_3$: $Z_p$ of `ACO` Heuristic (Chapter 4),
- $d_7$: $Z_p$ of `Zp-Hybrid` Heuristic (Chapter 5),
- $d_8$: $Z_p$ of `Zp-Estimator` Heuristic (Chapter 5).

## 6.1 Greedy Heuristic Output

`Greedy Heuristic` provides us with the improvement of $Z_p$; yet, we observe that in most cases, other heuristics produce better output compared to `Greedy Heuristic`. However, in the particular case of **cage11, cage12** and **e40r0100** matrices, `Greedy Heuristic` shows improvement, whereas the other heuristics have no effect on the $Z_p$ for those corresponding matrices. Therefore, on the edge cases, `Greedy Heuristic` can be a choice. The drawback of the heuristics is the consumed runtime, such as, for the aforementioned matrices, the runtimes (average on three different variants) in milliseconds (ms) are 666784, 7061533, and 98942, respectively.

## 6.2 ACO Superiority on Constructed Matrices

Based on the properties $(m, c_n, Z_{min}, Z_{max})$ of sparse matrices $abb313$, $ash331$, $af23560$, $impcol_e$ and $lp_maros_r7$, we crafted `Constructed Matrices` as described in table 6.2. The number of zeros in each row in `Constructed Matrices` is the same as their original counterpart, but appears as consecutive. The consecutive zeros for each row in `Constructed Matrices` start at a random column position. Following the creation of the matrix, the columns were randomly permuted to create test cases for our heuristics. In all the test

Table 6.1: Default parameters used in our ACO implementation

| Parameter | Default | Meaning / role |
|---|---|---|
| $n_{ants}$ | 25 | Number of ants. |
| $\alpha$ | 1 | Influence of pheromone $\tau_{ij}$ in the selection score. |
| $\beta$ | 2 | Influence of heuristic $\eta_{ij}$ (e.g., $1/(d_{ij}+\varepsilon)$). |
| $\rho$ | 0.5 | Global evaporation: $\tau \leftarrow (1-\rho)\tau$. |
| $\tau$ | $\frac{1}{\rho \cdot NN_{tour}}$ | For MMAS $\tau_{\max} = \frac{1}{\rho \cdot NN_{tour}}$, $\tau_{\min} = \frac{\tau_{\max}}{2 \cdot n}$, |
| $\varepsilon$ | $10^{-9}$ | Small constant to avoid division by zero in $\eta_{ij}$. |
| `nn_ants` | $\min(20, n-1)$ | Candidate list size for nearest-neighbor selection. |
| `ls_flag` | 3 | Local search (0: no local search, 1: 2-opt, 2: 2.5-opt, 3: 3-opt). |
| `dlb_flag` | $TRUE$ | Apply don't look bits in local search. |
| `nn_ls` | 20 | Use fixed radius for local search in specified number of nearest neighbours. |
| `max_tries` | 10 | Number of trial. |
| `max_tours` | 0 | Number of steps in each trial |
| `max_time` | 10 | Maximum time for each trial. |
| `optimal` | 1 | Stop if a tour better than or equal to the optimum is found. |
| `mmas_flag` | $TRUE$ | Enable MMAS variant |
| `seed` | time-based | Seed for the random number generator. |

cases, our ACO obtained a remarkable highest $Z_p$ improvement to $Z_{min}$. Additionally, we notice near-optimal performance from $Zp - Hyrbid$ Heuristics. These prove the effectiveness of framing the Consecutive Zeros Problem as a variant of TSP for the ACO, where pure distance and pheromone-based solution can reconstruct matrices with 100% accuracy. The additional $Z_p$ factor in $Zp - Hyrbid$ can be accounted for observed gap between the obtained result and the optimal $Z_p$. The default parameters utilized to run our ACO are illustrated in table 6.1.

Table 6.2: Test Results on `Constructed Matrices`, employing `ACO` ($d_3$), `Hybrid-Zp` ($d_7$) and `Zp-Estimator` ($d_8$)

| **Name** | **m** | **n** | $n_c$ | $Z_{min}$ | $Z_{max}$ | $d_0$ | $d_3$ | $d_7$ | $d_8$ |
|---|---|---|---|---|---|---|---|---|---|
| Abb313_1 | 313 | 176 | 10 | 4 | 9 | 1 | **4** | **4** | **4** |
| Abb313_2 | 313 | 176 | 10 | 4 | 9 | 1 | **4** | **4** | **4** |
| Abb313_3 | 313 | 176 | 10 | 4 | 9 | 2 | **4** | **4** | **4** |
| Abb313_4 | 313 | 176 | 10 | 4 | 9 | 2 | **4** | **4** | **4** |
| Abb313_5 | 313 | 176 | 10 | 4 | 9 | 2 | **4** | **4** | **4** |
| Ash331_1 | 331 | 104 | 6 | 4 | 4 | 1 | **4** | **4** | **4** |
| Ash331_2 | 331 | 104 | 6 | 4 | 4 | 2 | **4** | **4** | **4** |
| Ash331_3 | 331 | 104 | 6 | 4 | 4 | 2 | **4** | **4** | **4** |
| Ash331_4 | 331 | 104 | 6 | 4 | 4 | 1 | **4** | **4** | **4** |
| Ash331_5 | 331 | 104 | 6 | 4 | 4 | 1 | **4** | **4** | **4** |
| Af23560_1 | 23560 | 23560 | 41 | 20 | 30 | 2 | **20** | **20** | 8 |
| Af23560_2 | 23560 | 23560 | 41 | 20 | 30 | 3 | **20** | 19 | 8 |
| Af23560_3 | 23560 | 23560 | 41 | 20 | 30 | 3 | **20** | 18 | 7 |
| Af23560_4 | 23560 | 23560 | 41 | 20 | 30 | 3 | **20** | 19 | 8 |
| Af23560_5 | 23560 | 23560 | 41 | 20 | 30 | 3 | **20** | 19 | 8 |
| Impcol_e_1 | 225 | 225 | 21 | 9 | 20 | 2 | **9** | **9** | 5 |
| Impcol_e_2 | 225 | 225 | 21 | 9 | 20 | 2 | **9** | **9** | 4 |
| Impcol_e_3 | 225 | 225 | 21 | 9 | 20 | 2 | **9** | **9** | 5 |
| Impcol_e_4 | 225 | 225 | 21 | 9 | 20 | 2 | **9** | **9** | 5 |
| Impcol_e_5 | 225 | 225 | 21 | 9 | 20 | 2 | **9** | **9** | 4 |
| LP_maros_r7_1 | 3136 | 9408 | 83 | 35 | 78 | 3 | **35** | 34 | 15 |
| LP_maros_r7_2 | 3136 | 9408 | 83 | 35 | 78 | 3 | **35** | 34 | 15 |
| LP_maros_r7_3 | 3136 | 9408 | 83 | 35 | 78 | 3 | **35** | 34 | 15 |
| LP_maros_r7_4 | 3136 | 9408 | 83 | 35 | 78 | 3 | **35** | 34 | 15 |
| LP_maros_r7_5 | 3136 | 9408 | 83 | 35 | 78 | 3 | **35** | 34 | 15 |

## 6.3 Update Over Nearest Neighbor Heuristic

We test the performance of our heuristics on eligible matrices from [2] and compare with the Nearest Neighbor Heuristic and 2-opt algorithm. We noted a similar or higher performance employing our heuristics. The `ACO` and `Zp-Hybrid` heuristics yield higher $Z_p$ improvement on *impcol_e*, *lund_a*, and *lund_b*, however as discussed in section 6.1,

matrices *cage*11, *cage*12, and *e*40*r*0100 could only be improved by Greedy Heuristics.

Table 6.3: Comparison of Nearest Neighbor Heuristic and 2-opt algorithm with our heuristics

| **Name** | **m** | **n** | $\mathbf{n_c}$ | $\mathbf{Z_{min}}$ | $\mathbf{Z_{max}}$ | $\mathbf{d_0}$ | $\mathbf{d_1}$ | $\mathbf{d_2}$ | $\mathbf{d_3}$ | $\mathbf{d_7}$ | $\mathbf{d_8}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| abb313 | 313 | 176 | 10 | 4 | 9 | 1 | **2** | **2** | 1 | **2** | **2** |
| ash219 | 219 | 85 | 4 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| ash331 | 331 | 104 | 6 | 4 | 4 | **2** | **2** | **2** | **2** | **2** | **2** |
| ash608 | 608 | 188 | 6 | 4 | 4 | **2** | **2** | **2** | **2** | **2** | **2** |
| ash958 | 958 | 292 | 6 | 4 | 4 | **2** | **2** | **2** | **2** | **2** | **2** |
| af23560 | 23560 | 23560 | 41 | 20 | 30 | 2 | **3** | **3** | **3** | **3** | **3** |
| cage11 | 39082 | 39082 | 62 | 31 | 59 | **4** | **4** | **4** | **4** | **4** | **4** |
| cage12 | 130228 | 130228 | 68 | 35 | 63 | **4** | **4** | **4** | **4** | **4** | **4** |
| e40r0100 | 17281 | 17281 | 69 | 7 | 61 | **1** | **1** | **1** | **1** | **1** | **1** |
| fs_541_1 | 541 | 541 | 13 | 2 | 12 | **1** | **1** | **1** | **1** | **1** | **1** |
| fs_541_2 | 541 | 541 | 13 | 2 | 12 | **1** | **1** | **1** | **1** | **1** | **1** |
| fs_541_3 | 541 | 541 | 13 | 2 | 12 | **1** | **1** | **1** | **1** | **1** | **1** |
| impcol_b | 59 | 59 | 11 | 4 | 9 | **2** | **2** | **2** | **2** | **2** | **2** |
| impcol_e | 225 | 225 | 21 | 9 | 20 | 1 | 4 | 4 | 4 | **5** | 4 |
| lp_ken_11 | 14694 | 21349 | 125 | 3 | 124 | **2** | **2** | **2** | **2** | **2** | **2** |
| lp_maros_r7 | 3136 | 9408 | 83 | 35 | 78 | **3** | **3** | **3** | **3** | **3** | **3** |
| lund_a | 147 | 147 | 24 | 12 | 23 | 3 | 1 | 1 | **5** | **5** | 4 |
| lund_b | 147 | 147 | 24 | 12 | 23 | 3 | 1 | 1 | 4 | **6** | 4 |

## 6.4 Overall Comparison of Heuristics

We present the extensive test results of our work in this section. The findings reveal the overall superiority of `Zp-Hybrid` heuristic over the remaining approaches. The results further indicate that the `Zp-Estimator` performs comparably to, or surpasses, the `ACO` heuristic in most instances. Most of these real-life instances we tested could be improved, indicating the effectiveness of our implementation. Therefore, the employment of the heuristics could achieve greater computational advantages while calculating sparse Jacobian matrices.

Table 6.4: Overall performance of the heuristics

| **Name** | **m** | **n** | $\mathbf{n_c}$ | $\mathbf{Z_{min}}$ | $\mathbf{Z_{max}}$ | $\mathbf{d_0}$ | $\mathbf{d_3}$ | $\mathbf{d_7}$ | $\mathbf{d_8}$ |
|---|---|---|---|---|---|---|---|---|---|
| sherman1 | 1000 | 1000 | 8 | 4 | 7 | 1 | 1 | **2** | **2** |
| e30r2000 | 9661 | 9661 | 65 | 3 | 57 | **1** | **1** | **1** | **1** |
| lpi_reactor | 318 | 808 | 78 | 12 | 76 | 1 | 5 | **11** | 7 |
| sphere3 | 258 | 258 | 10 | 4 | 9 | 1 | **2** | **2** | **2** |
| bcsstk20 | 485 | 485 | 11 | 4 | 10 | 2 | 2 | **3** | 2 |
| bcsstm07 | 420 | 420 | 28 | 12 | 27 | 3 | 3 | **4** | 3 |
| dwt_221 | 221 | 221 | 12 | 3 | 11 | **2** | **2** | **2** | **2** |
| dwt_918 | 918 | 918 | 14 | 5 | 13 | 2 | 2 | **3** | **3** |
| mesh2e1 | 306 | 306 | 10 | 3 | 9 | **1** | **1** | **1** | **1** |
| nos3 | 960 | 960 | 18 | 8 | 17 | **6** | **6** | **6** | **6** |
| nos5 | 468 | 468 | 25 | 10 | 24 | 3 | 3 | **4** | **4** |
| nos6 | 675 | 675 | 5 | 2 | 4 | 1 | **2** | **2** | **2** |
| GL6_D_9 | 340 | 545 | 31 | 3 | 31 | 1 | 1 | **2** | **2** |
| mk9-b1 | 378 | 36 | 7 | 5 | 5 | **3** | **3** | **3** | **3** |
| n3c5-b5 | 210 | 252 | 11 | 5 | 5 | 1 | 1 | **2** | **2** |
| n3c5-b6 | 120 | 210 | 11 | 4 | 4 | **1** | **1** | **1** | **1** |
| robot24c1_mat5 | 404 | 302 | 102 | 11 | 99 | 3 | 3 | **8** | 5 |
| gre_512 | 512 | 512 | 8 | 3 | 6 | **1** | **1** | **1** | **1** |
| jagmesh1 | 936 | 936 | 9 | 2 | 8 | **1** | **1** | **1** | **1** |
| jagmesh5 | 1180 | 1180 | 9 | 2 | 8 | **1** | **1** | **1** | **1** |
| plat1919 | 1919 | 1919 | 24 | 9 | 23 | 1 | **2** | **2** | **2** |
| steam1 | 240 | 240 | 24 | 3 | 15 | **3** | **3** | **3** | 2 |
| can_256 | 256 | 256 | 83 | 30 | 82 | 16 | 14 | **24** | 23 |
| can_268 | 268 | 268 | 37 | 4 | 36 | **4** | **4** | **4** | **4** |
| can_292 | 292 | 292 | 35 | 3 | 34 | 2 | **3** | **3** | 2 |
| can_634 | 634 | 634 | 28 | 7 | 27 | 2 | 2 | **4** | 3 |
| can_715 | 715 | 715 | 105 | 63 | 104 | 25 | 29 | **32** | 30 |
| can_1054 | 1054 | 1054 | 35 | 23 | 34 | 5 | 6 | **7** | 6 |
| can_1072 | 1072 | 1072 | 35 | 22 | 34 | 5 | 6 | **7** | 6 |
| west0067 | 67 | 67 | 8 | 2 | 7 | **1** | **1** | **1** | **1** |
| west0381 | 381 | 381 | 28 | 3 | 27 | 2 | 1 | **2** | **2** |
| lp_stair_T | 614 | 356 | 36 | 2 | 35 | **2** | **2** | **2** | **2** |

# Chapter 7

# Conclusion and Future Works

In this thesis, we included a detailed study on Ant Colony optimization and illustrated the novel approach (to our knowledge) of using this powerful heuristic to solve the Consecutive Zeros problem. We introduced `Greedy Heuristics`, along with two Zp-oriented variants of the `ACO`, `Zp-Hybrid` and `Zp-Estimator`, which deliver promising results. Moreover, we exemplified the space optimization of the implementation using arrays and a corresponding array index calculation procedure, eliminating the need to store data in matrices. We have discussed all the pertinent background concepts and connected all the concepts to demonstrate the sequence of operations required to reconstruct a Jacobian matrix. The step-by-step workflow can be instrumental for future researchers. As the problem we are trying to solve is NP-hard by nature, the proposed heuristics provide an efficient technique to get closer to optimal, and reach optimal in some cases.

## 7.1 Future Research Directions

At present, we are implementing $Z_p - Hybrid$ and $Z_p - Estimator$ as two separate entities; however, incorporating both into a single heuristic where factors unique to the heuristics are combined with distance and pheromone-based approach, could be a potential avenue for future investigation. The optimization of Greedy Heuristics time complexity can be a fruitful research area. Fusing the solutions in this thesis with the `DSJM` toolkit to automate the final optimized coloring and compression of the Jacobian could be a prospective and exciting research trajectory. The extensive analytical study on the effects of different

ACO variants (AC, EAS, RAS, BWAS, ACS, MMAS, etc) on solving the Consecutive Zeros Problem is also an interesting research direction. Moreover, the inclusion and impact of local_search with these variants could be an exciting study.

# Bibliography


[1] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010.

[2] B V Ravi Chandra and Shahadat Hossain. Computing sparse derivatives and consecutive zeros problem. *Journal of Physics: Conference Series*, 410(1):012051, feb 2013.

[3] A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse jacobian matrices. *Journal of the Institute of Mathematics and its Applications*, 13(1):117–120, 1974.

[4] Timothy A. Davis and Yifan Hu. Suitesparse matrix collection: Hb/bcsstm26. `https://sparse.tamu.edu/HB/bcsstm26`, 2011. Accessed 2025-10-16.

[5] M. Dorigo and G. Di Caro. Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 2, pages 1470–1477 Vol. 2, 1999.

[6] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE transactions on systems, man, and cybernetics, part b (cybernetics)*, 26(1):29–41, 1996.

[7] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. MIT Press, Cambridge, MA, 2004.

[8] Sardar Anisul Haque, Shahadat Hossain, and Marc Moreno Maza. Cache friendly sparse matrix-vector multiplication. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, pages 175–176, 2010.

[9] Mahmudul Hasan, Shahadat Hossain, Ahamad Imtiaz Khan, Nasrin Hakim Mithila, and Ashraful Huq Suny. Dsjm: A software toolkit for direct determination of sparse jacobian matrices. In Gert-Martin Greuel, Thorsten Koch, Peter Paule, and Andrew Sommese, editors, *Mathematical Software – ICMS 2016*, pages 275–283, Cham, 2016. Springer International Publishing.

[10] Shahadat Hossain and Ahamad I. Khan. Exact coloring of sparse matrices. In D. Marc Kilgour, Herb Kunze, Roman Makarov, Roderick Melnik, and Xu Wang, editors, *Recent Advances in Mathematical and Statistical Methods*, pages 23–36, Cham, 2018. Springer International Publishing.

[11] Shahadat Hossain and Trond Steihaug. *Reducing the Number of AD Passes for Computing a Sparse Jacobian Matrix*, pages 263–270. Springer New York, New York, NY, 2002.

[12] Shahadat Hossain and Trond Steihaug. Sparsity issues in the computation of jacobian matrices. In *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, ISSAC '02, page 123–130, New York, NY, USA, 2002. Association for Computing Machinery.

[13] Shahadat Hossain and Trond Steihaug. Optimal direct determination of sparse jacobian matrices. *Optimization Methods and Software*, 2012.

[14] Shahadat Hossain and Trond Steihaug. Graph models and their efficient implementation for sparse jacobian matrix determination. *Discrete Applied Mathematics*, 161(12):1747–1754, 2013. 9th Cologne/Twente Workshop on Graphs and Combinatorial Optimization (CTW 2010).

[15] Shahadat Hossain and Trond Steihaug. Sparse matrix computations with application to solve system of nonlinear equations. *Wiley Interdisciplinary Reviews: Computational Statistics*, 5:372–386, 2013.

[16] Timothy A. Davis. The SuiteSparse Matrix Collection. `https://sparse.tamu.edu/`, 2025. Accessed: 2025-10-22.