UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS – CCT
PROGRAMA DE PÓS-GRADUAÇÃO – PPGCAP

RAFAEL GRANZA DE MELLO

CATEGORISED GROUPING: A FRAMEWORK FOR INDUSTRY APPLICATIONS

JOINVILLE
2025

RAFAEL GRANZA DE MELLO

CATEGORISED GROUPING: A FRAMEWORK FOR INDUSTRY APPLICATIONS

Master thesis submitted to the Computer Science Department at the College of Technological Science of Santa Catarina State University in fulfillment of the partial requirement for the Master's degree in Applied Computing.

Supervisor: Yuri Kaszubowski Lopes
Co-supervisor: Rui Jorge Tramontin Junior

JOINVILLE
2025

Ficha catalográfica elaborada pelo programa de geração automática da

Biblioteca Universitária Udesc,

com os dados fornecidos pelo(a) autor(a)

Mello, Rafael Granza de
Categorised grouping : a framework for industry applications / Rafael Granza de Mello. -- 2025.
117 p.

Orientador: Yuri Kaszubowski Lopes
Coorientador: Rui Jorge Tramontin Junior
Dissertação (mestrado) -- Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada, Joinville, 2025.

1. Matching. 2. Agrupamento. 3. Algoritmo. 4. Framework. 5. Otimização. I. Lopes, Yuri Kaszubowski. II. Tramontin Junior, Rui Jorge. III. Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada. IV. Titulo.

RAFAEL GRANZA DE MELLO

CATEGORISED GROUPING: A FRAMEWORK FOR INDUSTRY APPLICATIONS

Master thesis submitted to the Computer Science Department at the College of Technological Science of Santa Catarina State University in fulfillment of the partial requirement for the Master's degree in Applied Computing.

Supervisor: Yuri Kaszubowski Lopes
Co-supervisor: Rui Jorge Tramontin Junior

EXAMINING COMMITTEE:

Yuri Kaszubowski Lopes PhD.
Universidade do Estado de Santa Catarina

Rui Jorge Tramontim Junior PhD.
Universidade do Estado de Santa Catarina

Members:

Ricardo José Pfitscher PhD.
Universidade Federal de Santa Catarina

Gilmário Barbosa do Santos PhD.
Universidade do Estado de Santa Catarina

Fabiano Baldo PhD.
Universidade do Estado de Santa Catarina

Joinville, July 14, 2025

To my family, for putting up with me during this period. And to my friends, for putting up with me after it!

ACKNOWLEDGEMENTS

I would like to thank my advisor for accepting to guide my research work.

To all my professors at the Universidade do Estado de Santa Catarina – UDESC, for their excellence and technical expertise.

To my parents, who have always been by my side, supporting me throughout my entire journey. I am grateful to Júlia and my family for the unwavering support they have given me throughout my life.

A special thank you to my advisor for the encouragement and for dedicating his scarce time to my research project.

ABSTRACT

Matching problems are critical in various industrial applications, such as task allocation, scheduling, and resource distribution. However, existing optimization solutions are often complex, rigid, or inaccessible to professionals without specialized expertise. To mitigate this issue, this dissertation proposes the design of a flexible model to define and solve matching and grouping problems, enabling users to configure relational data and optimization constraints without requiring deep knowledge of optimization techniques. The proposed method operates on the application's ORM (Object-Relational Mapping) model, which enhances the method's adoption given the current widespread use of ORM in the industry. In addition to the model design, this work presents a prototype implementation and introduces an innovative matching algorithm under quota constraints. This algorithm ensures fairness and feasibility in resource allocation scenarios where quotas play a crucial role. By bridging the gap between theoretical matching models and their practical applications, this research offers a structured yet adaptable approach to solving real-world matching problems. **Keywords**: Matching. Grouping. Algorithm. Framework. Optimization.

RESUMO

Os problemas de *matching* são críticos em diversas aplicações industriais, como alocação de tarefas, agendamento e distribuição de recursos. No entanto, as soluções de otimização existentes são frequentemente complexas, rígidas ou inacessíveis para profissionais sem experiência especializada. Prara mitigar este problema, esta dissertação propõe o design de um modelo flexível para definir e resolver problemas de *matching* e agrupamento, permitindo que os usuários configurem dados relacionais e restrições de otimização sem a necessidade de um conhecimento profundo em técnicas de otimização. O método proposto opera sobre o modelo OMR (Mapeamento Relacional de Objetos, do Inglês *Object-Relational Mapping*) da aplicação, o que potencializa a adoção do método, haja visto a atual adoção do ORM pela indústria. Além do design de um modelo, este trabalho apresenta um protótipo de uma implementação e introduz um algoritmo de matching inovador sob restrições de cotas. Esse algoritmo garante a equidade e a viabilidade em cenários de alocação de recursos onde as cotas desempenham um papel crucial. Ao preencher a lacuna entre os modelos teóricos de matching e suas aplicações práticas, esta pesquisa oferece uma abordagem estruturada, porém adaptável, para resolver problemas reais de *matching*. **Palavras-chave**: Matching. Agrupamento. Algoritmo. *Framework*. Otimização.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AI	Artificial Inteligence
GNN	Graph Neural Networks
MCMF	Minimum Cost Maximum Flow
NLP	Natural Language Processing
ORM	Object Relational Mapping
SCF	Set Coverage Framework
UDESC	Universidade do Estado de Santa Catarina

CONTENTS

APPENDIX A – GROUPRULE AND GROUP CLASS . . . 98
APPENDIX B – HUNGARIAN ALGORITHM SOLVER . . . 101

1 INTRODUCTION

Matching problems are fundamental in various real-world applications, including job and task assignment, scheduling, resource distribution, and recommendation systems. The efficiency of solving such problems directly impacts operational performance in multiple industries, from logistics and healthcare to finance and human resources. Despite their significance, many matching problems are computationally complex, often requiring advanced optimization techniques.

Existing approaches typically rely on specialized optimization frameworks or domain-specific heuristics [1]. While these methods can be effective, they present challenges regarding accessibility and adaptability. Many existing frameworks require extensive expertise in combinatorial optimization, making them difficult to adopt for professionals without specialized knowledge. Additionally, rigid modeling approaches may not generalize well to different industry needs, limiting their applicability.

Given these challenges, this dissertation proposes a flexible and accessible framework for solving various matching problems. The framework leverages relational data and user-defined optimization constraints, allowing for intuitive problem formulation while maintaining computational efficiency. By bridging the gap between theoretical optimization techniques and practical implementation, the proposed framework seeks to provide an effective tool for a broad audience of developers and industry professionals.

1.1 MOTIVATION

Many matching problems belong to the class of NP-hard problems (see [2]), requiring sophisticated algorithms to obtain near-optimal solutions within feasible time constraints. However, the practical adoption of these techniques is hindered by the lack of accessible tools that can be adapted to different industrial contexts. Existing solutions often require dedicated teams of experts, increasing implementation costs and reducing scalability.

By developing a framework that simplifies the formulation and resolution of matching problems, this work aims to democratize access to advanced optimization techniques. The proposed approach enables professionals to efficiently configure and apply matching algorithms, reducing dependency on specialized expertise and facilitating adoption across diverse sectors.

1.2 RESEARCH OBJECTIVES

The primary objective of this research is to develop a comprehensive framework capable of addressing a broad range of matching problems through configurable optimization strategies. Specifically, this work has the following Research Objectives (RO):

RO1. Design an adaptable framework that supports various types of matching problems, including one-to-one, many-to-many, and constrained matchings.

RO2. Integrate in the framework different solvers, including exact algorithms and heuristics, to balance computational efficiency and solution quality.

RO3. Provide an intuitive interface for defining problem constraints, allowing users to customize matching models without requiring deep expertise in optimization.

RO4. Introduce a novel approach for solving matching problems under fairness constraints, ensuring compliance with minimum quota requirements.

1.3 STRUCTURE OF THE DISSERTATION

The remainder of this dissertation is structured as follows. Chapter 2 presents fundamental concepts in matching problems, graph theory, optimization techniques, and relational diagram usage. Chapter 3 reviews related work, highlighting existing algorithms and new tendencies. Chapter 4 describes the proposed framework's architecture and design principles. Chapter 5 presents an algorithm for matching under minimum quotas. Chapter 6 details a preliminary implementation, illustrating the framework's adaptability through real-world applications. Finally, Chapter 7 summarizes the contributions, discusses limitations, and outlines directions for future research.

2 BACKGROUND

In order to understand and construct a framework capable of solving a wide variety of matching problems, it is essential to first explore the theoretical foundations that support this approach. This chapter presents the key concepts necessary for modeling, analyzing, and solving grouping problems in industry.

We begin by introducing basic elements from graph theory, which is the foundation for representing entities and relationships in matching problems. Then, we explore the formal concept of matching, including fairness constraints and cost optimization using flow networks. To complement this, we review the set cover problem, which offers a broader perspective on grouping scenarios that go beyond pairwise relations.

As the proposed solution is designed to be adaptable to real-world software systems, we also introduce the concept of Object-Relational Mapping (ORM), a widely used modeling paradigm in the software industry. This allows the framework to be integrated directly with relational data models in existing applications.

Finally, we present an overview of genetic algorithms, a class of metaheuristics suitable for solving complex optimization problems where exact methods may be infeasible. These algorithms will later be applied as solvers within the proposed framework.

Together, these concepts provide the theoretical and practical basis for the design and implementation of the categorised grouping framework developed in this dissertation.

2.1 GRAPHS

Graphs are a fundamental concept in computer science and mathematics, widely used to represent relationships between objects. A graph G is defined as a pair (V,E), where V is a set of vertices (or nodes) and E is a set of edges [3]. Each edge in E connects a pair of vertices.

2.1.1 Vertices and Edges

The primary components of a graph are vertices and edges [3, 4]. A vertex (singular of vertices) represents a discrete entity in the graph, often denoted as $v \in V$. An edge connects two vertices and represents the relationship between them. An edge is typically denoted as $e = (u,v) \in E$, where $u,v \in V$. In Figure 1, the vertices are represented as circles and the edges are represented as lines.

2.1.2 Subgraphs

A subgraph G' of a graph G is a graph formed from a subset of the vertices and edges of G [3, 4]. Formally, $G' = (V',E')$ is a subgraph of $G = (V,E)$ if $V' \subseteq V$ and $E' \subseteq E$. Figure 2 provide examples of subgraphs of Graph G presented in Figure 1.

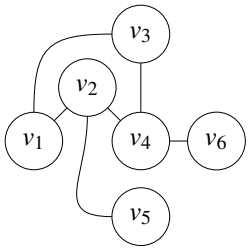
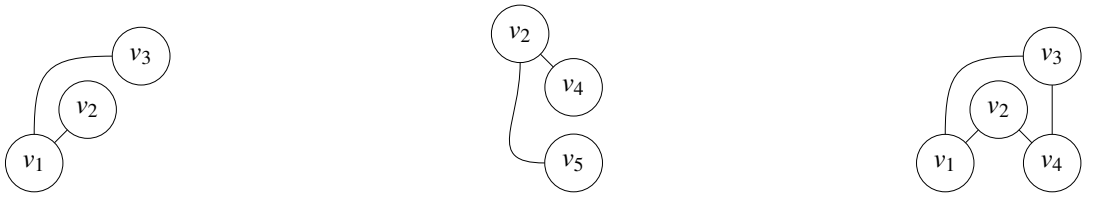


Figure 1 – Graph $G = (V, E)$. The circles $(v_1, \ldots, v_6)$ are the vertices V; the lines connecting the vertices are the edges E.



(a) Subgraph G_1 (b) Subgraph G_2 (c) Subgraph G_3

Figure 2 – Three possible subgraphs of Graph G (See Figure 1).

2.1.3 Subsets of Vertices

Subsets of vertices play a crucial role in various graph algorithms and properties [3, 4]. Given a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ can be used to define induced subgraphs, vertex covers, and other structures. For example, the induced subgraph $G[S]$ is formed by the vertices in S and all the edges between them in G. Figure 3 shows a subset of vertices of G_2 (See Figure 2b).

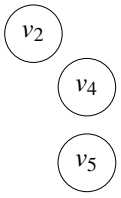


Figure 3 – Subset of vertices from Graph G (Figure 1) which induces (generates) the Graph G'' (Figure 2b).

2.1.4 Bipartite Graphs

A bipartite graph is a graph $G = (V, E)$ whose vertices can be divided into two disjoint sets V_1 and V_2 such that every edge in E connects a vertex in V_1 to a vertex in V_2 [3, 4]. Formally, a graph G is bipartite if V can be partitioned into two sets V_1 and V_2 such that $V_1 \cap V_2 = \emptyset$ and every edge in E has one endpoint in V_1 and the other in V_2. Figure 4c depics examples of bipartite graphs and Figure 5 shows an example of a non-bipartite graph.

Bipartite graphs have many applications in various fields, including computer science, biology, and social sciences. They are used to model relationships between two different classes of objects, such as students and courses, users and items, or proteins and genes. One common

algorithmic problem on bipartite graphs is the maximum bipartite matching problem, which aims to find the largest subset of edges in the graph such that no two edges share a common vertex.

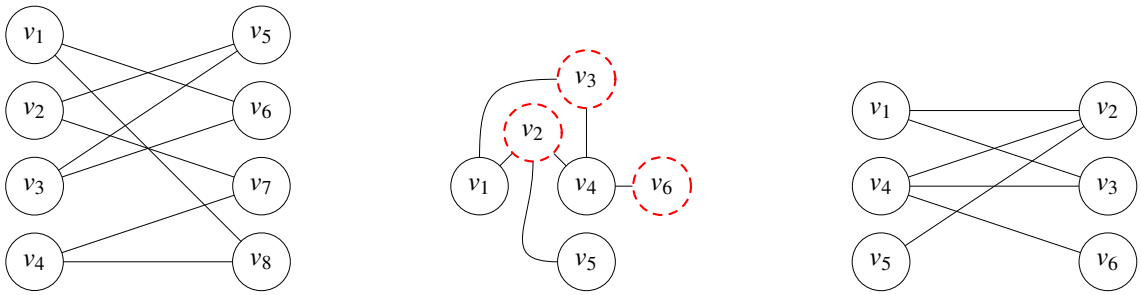


Figure 4 – Examples of bipartite graphs and ways to represent it. (a) Common representation of a bipartite graph (by layer). (b-c) the same Graph G (See Figure 1), with different bipartite representations. (b) Representation by color, black / solid nodes represent one partition and red / dashed nodes represent the other partition. (c) Representation by layer.

It is important to note that not every graph is bipartite. A graph that contains an odd-length cycle cannot be bipartite, as it is impossible to partition the vertices into two sets without having two vertices of the same set being adjacent. For example, consider the following graph:

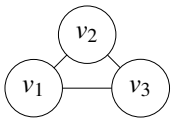


Figure 5 – Example of a non-bipartite graph.

This graph contains a cycle of length 3 (vertices v_1, v_2, and v_3), which is an odd-length cycle. Therefore, it is not possible to divide the vertices into two sets where all edges connect vertices from different sets, making this graph non-bipartite.

2.2 MATCHING

A matching (strictly in the context of graph theory) in a graph $G = (V,E)$ is a set of edges $M \subseteq E$ such that no two edges in M share a common vertex [3, 4]. Matchings are used to model relationships in various applications, such as assigning tasks to workers or pairs in social networks. Figure 6 shows a possible matching of Graph G (see Figure 1).

In a bipartite graph, a matching (also called bipartite matching) is a subset of edges $M \subseteq E$ such that each vertex is incident to at most one edge in M. Finding a maximum matching in a bipartite graph can be efficiently solved using algorithms like the Hopcroft-Karp algorithm [5, 6].

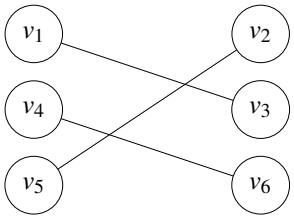


Figure 6 – Possible matching of Graph G (see Figure 1).

2.2.1 Bipartite Matching

Bipartite Matching, also known as bipartite pairing is a fundamental problem in graph theory, with many practical applications, including resource allocation, market design, and job assignment. We have two distinct sets of elements, and the objective is to find corresponding pairs between them while also optimizing some criteria.

To formalize the problem, consider a graph $G = (V, E)$, where the vertex set V is partitioned into two disjoint subsets V_1 and V_2, such that $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$. The goal is to identify a set of edges $M \subseteq E$ that connect vertices from V_1 to V_2, ensuring that no two edges in M share a common vertex. This configuration is commonly referred to as a *matching* [7].

It is important to note that in some cases, sets have different sizes, and thus it is not possible to match every vertex of both sets.

2.2.2 Fairness in Matching

Fairness in matching consists in take into account specific attributes of the set of vertices and their proportionality in the solution. Inspired by the work of [8], the notion of fairness adopted by this study is delegated to an external operator. This impartial operator does not directly engage with the sets being matched. Instead, it has the responsibility to judge the fairness in the matching context.

In the search for pairing a set U and a set V, the concept of giving priority to some specific subset of U that contains specific traits is proposed. This different approach to the original problem is particularly relevant in scenarios where prioritizing different characteristics alongside minimal cost is desired.

To operationalize this prioritization, this work employs the concept of minimum quotas, which establish a minimum number of vacancies reserved for specific subsets. These minimum quotas, act as instruments to promote equity in the matching process, ensuring those specific subsets receive adequate representation.

The quotas refer to the predetermined allocation of a minimum number of matchings to each special subset, thereby establishing a fair distribution of pairings. This definition is crucial for ensuring representativity and preventing imbalances in the allocations.

It is important to highlight that, in this study, the quotas are only applied to one set of the bipartite graph, meaning that only U or V is involved in the quota implementation.

This definition of quotas aligns with the fairness proposed by [8], where the imposition

of quotas becomes essential to maximize equity in the resources allocated. The next session approaches how this definition of fairness is represented in the mapping to the MCMF problem.

2.2.3 Minimum Cost Maximum Flow

The minimum cost maximum flow problem refers to the search for the maximum amount of possible flow in a network, considering costs associated with the passage of flow through certain uniderected edges [9, 10]. In other words, the objective of such problems is to optimize the transportation of resources from one point to another while minimizing the costs involved in this process.

This concept has broad applications, often used in transportation problems, network design, and linear programming, among others. Efficient resolution of Minimum-Cost Maximum Flow (MCMF) problems is crucial in various fields, contributing to the efficiency and economy of resources.

2.2.4 Solving Bipartite Matching with MCMF

To clarify the methodology of mapping a Bipartite Matching problem to a Minimum-Cost Maximum Flow (MCMF) problem, a method that is already well-known and documented [11, 12, 13], we present a detailed visual representation of this process in Figure 7. In this figure, the bipartite graph being matched is the set of nodes $U = \{u_1, \ldots, u_6\}$ (in red/dashed) and $V = \{v_1, \ldots, v_3\}$ (in blue/dotted). The edges connecting the sets U and V have two parameters $(K;C)$: the first parameter K, denotes the capacity, indicating the maximum number of flows that can be assigned to a given task, while the second parameter C, represents the cost associated with the edge.

The mapping strategy involves representing the bipartite graph as a flow network. Two other nodes are added, a Source, connected to all nodes in Set U, and a Sink, connected to all nodes in Set V. It is worth noting that the edges connected to the Source and Sink sets have zero cost and unit capacity $(1;0)$ to guarantee that the added connections do not affect the overall cost minimization.

In this context, the vertices are represented by the sets U and V, while the edges define the relationship between them, each associated with a specific cost. Each edge connecting U to V carries a cost, which may represent, for instance in a job allocation problem, the hiring cost. The algorithm seeks to optimize this matching by minimizing the total cost, which can reflect, in practical terms, the reduction of operational or resource allocation expenses.

The MCMF algorithm is then applied to this network representation to determine the optimal allocation. For the problem instance presented in Figure 7, the result of the application of the MCMF algorithm generates the optimal flow (matching) displayed in Figure 8. In Figure 8, the selected nodes are highlighted with full colors, forming the match: $\{(u_1, v_1), (u_2, v_2), (u_4, v_3)\}$. Note that multiple optimal flows may be possible.

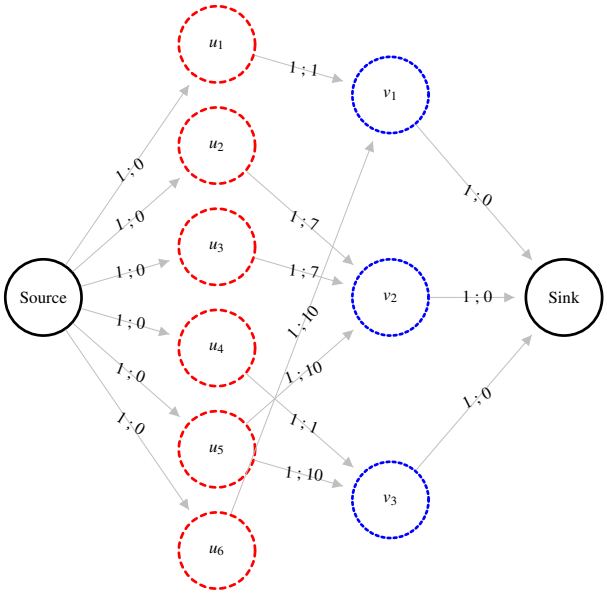


Figure 7 – Mapping of bipartite matching to flow problem.

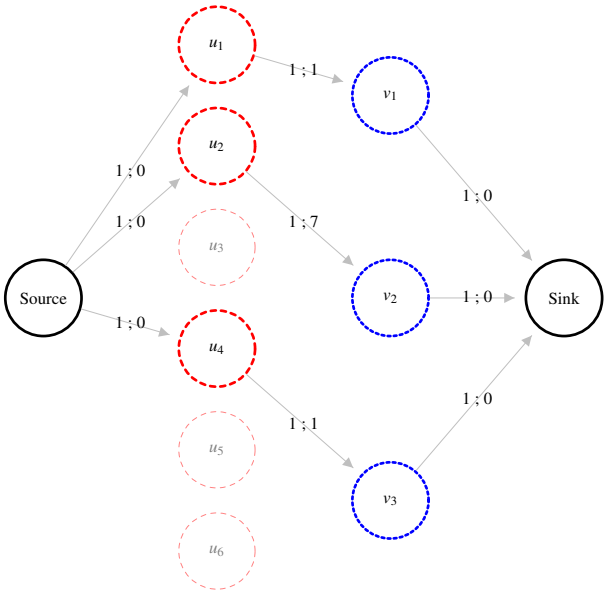


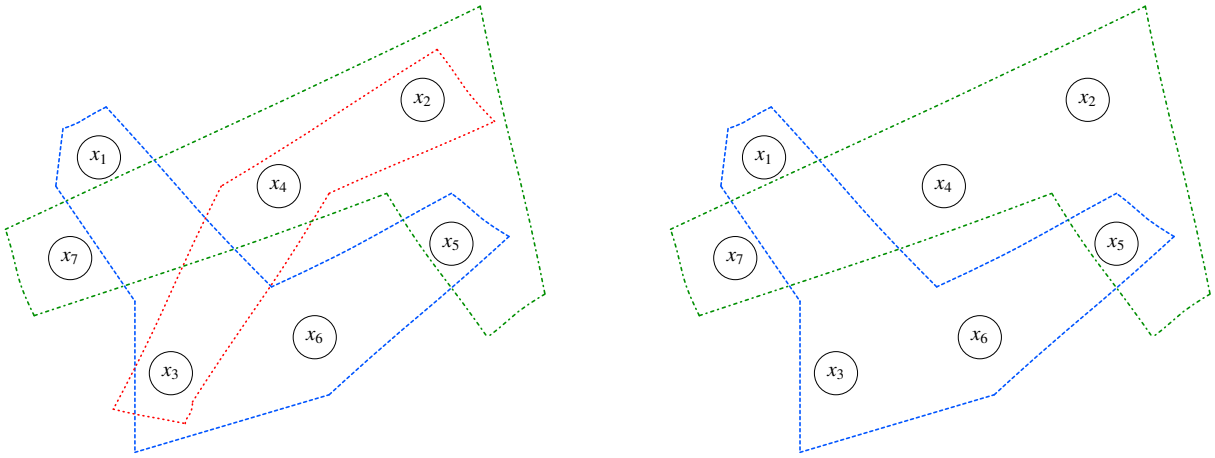
Figure 8 – Solution of the mapping of bipartite matching to flow problem. The minimum cost of the matching is defined as the sum of the costs of the selected edges, which is 9 in this case.

By exploring this approach, we can leverage theoretical advancements in MCMF algorithms, such as their use in parallelized processors [14], near-linear time algorithms [15], decentralized network computation [16], and potentially quantum algorithms [17].

2.3 SET COVER

Set cover is a classic problem in computer science and mathematics. Given a set X of elements and a collection S of subsets of X, the set cover problem is to find the smallest subcollection of S whose union is equal to X[3, 4]. It has applications in scheduling, DNA

sequencing, and data compression, and is one of Karp's 21 NP-complete problems [2]. Figure 9 shows two examples of Set Cover, where both examples cover all elements in X. However, the first example is redundant, as it uses subsets S_1, S_2, and S_3, while the second example is optimal, as it uses only subsets S_1 and S_2 to cover all elements in X.



(a) A Non-Optimal Set Cover. Subsets selected are s_1, s_2, and s_3.

(b) A Optimal Set Cover. Subsets selected are s_1 and s_2.

Figure 9 – Example of Set Cover. Where $S = \{s_1, s_2, s_3\}$, $s_1 = \{x_1, x_3, x_6, x_5\}$, $s_2 = \{x_7, x_4, x_2, x_5\}$ and $s_3 = \{x_3, x_4, x_2\}$.

2.3.1 Exact Set Cover

The exact set cover problem is a variant where each element in X must be covered exactly once by the selected sets. This variant is particularly relevant in scenarios where overlapping cover is not allowed or desired, such as certain types of resource allocation [18] and exact sequence alignment in computational biology [19]. Finding an exact set cover is also an NP-complete problem [2], requiring sophisticated algorithms or approximation techniques for large instances.

2.3.2 Weighted Set Cover

In the weighted set cover problem, each set in the collection S is assigned a positive weight, which represents its cost. The objective is to find a set cover that minimizes the total weight. The unweighted version of the problem can be understood as a particular case in which all sets in S have the same weight, typically equal to 1 [20].

Formally, let X be a set of n elements and let $S = \{S_1, S_2, \ldots, S_m\}$ be a collection of m subsets of X. Each subset S_i is associated with a weight w_i. The objective is to find a sub-collection $C \subseteq S$ such that the union of all sets in C equals X, and the total weight, defined as the sum of the weights of the sets in C, is minimized.

This problem appears in several practical situations. In network design, for instance, different network configurations may have different costs [21], and the goal is to ensure that all

network nodes are covered while minimizing the overall cost. In the context of sensor placement, each sensor may have a specific deployment cost [22], and the aim is to cover a given region using the least expensive combination of sensors. Another relevant example is the timing table problem, where tasks must be scheduled within a predefined timeline [18]. Each task may have a different cost associated with its execution time, and the objective is to schedule all required tasks while minimizing the total cost.

As with the unweighted version, the weighted set cover problem is classified as NP-hard [23]. This implies that there is no known algorithm capable of solving all instances of the problem exactly in polynomial time. Nonetheless, there are approximation algorithms that provide solutions close to the optimal within a feasible time frame, even for large problem instances [20].

The set cover problem, whether in its weighted or unweighted form, is a fundamental topic in theoretical computer science. It has inspired the development of many algorithmic techniques, such as greedy strategies [20], linear programming relaxations [24], and randomized methods [25]. A thorough understanding of this problem is crucial for addressing more complex optimization challenges in diverse domains.

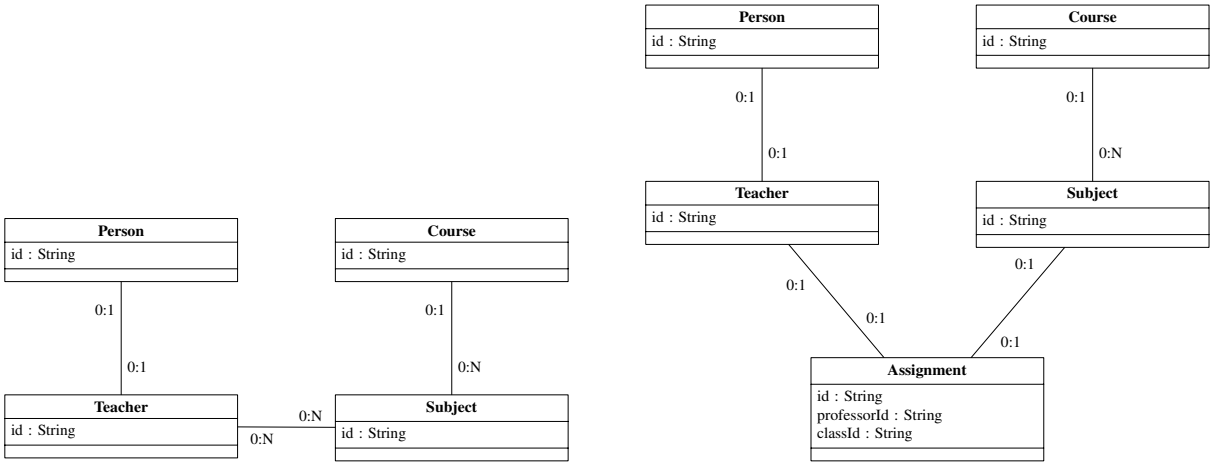
2.4 ORM (OBJECT-RELATIONAL MAPPING)

ORM (Object-Relational Mapping) is a programming technique that allows developers to map objects from object-oriented programming languages to relational database tables [26]. This technique facilitates the integration between the object-oriented and relational paradigms by automating the conversion of data between incompatible systems. In ORM, each class in an object-oriented system becomes a table/entity in a relational database, and the attributes of the class correspond to the columns of the table.

Relationships between classes are also represented in ORM [27]. For example, a one-to-many relationship between two classes would be represented by a foreign key in the database table of the "many" class, pointing to the primary key of the "one" class. This mapping ensures that the relational database structure reflects the object-oriented design of the application. Figure 10a demonstrates these relations, with examples of one-to-one, many-to-one and many-to-many relations. In order to implement a many-to-many relation, a relation class is created, as shown in Figures 10a and 10b.

ORM is commonly used in industry to simplify the development process and reduce the amount of boilerplate code needed for database interactions. By abstracting away the complexities of SQL queries and database schema management, ORM frameworks enable developers to focus more on the business logic of the application and less on the intricacies of database operations [28].

There are various ORM frameworks and tools available for different programming languages and databases. Some popular ORM frameworks include Hibernate for Java, Entity



(a) Representation of simple relations between entities of a school.

(b) New Class *Assignment* to represent the many-to-many relation of Figure 10a.

Figure 10 – Examples of implicit relations and their representation in UML.

Framework for .NET, and Django for Python [29]. These frameworks provide robust mechanisms for data manipulation, transaction management, and query generation, thus enhancing productivity and ensuring consistency in database interactions.

2.4.1 Relation Rules

In the context of object-relational mapping (ORM), the relation rules (relation geometry) describes the structure and constraints that govern relationships between entities in a database schema. It encompasses the rules defining how entities interact, how many objects can relate to one another, and how these relationships are navigated. Understanding this geometry is essential for designing consistent and efficient database systems that accurately reflect the domain's requirements.

2.4.1.1 One-to-One (1:1) Relationships

A **one-to-one relationship** represents a scenario where each instance of an entity is related to exactly one instance of another entity and vice versa. This type of relationship is typically used when an entity's attributes are logically split into separate tables for modularity or privacy purposes.

For example, in a system where each individual has one unique passport, the relationship between the entities Person and Passport can be modeled as one-to-one. In ORM terms, the Person entity includes a foreign key that links to the Passport entity, and the reverse link is also maintained. The cardinality constraints for this relationship are:

- 0 : 1: An optional relationship where an entity might not be linked (e.g., a person without a passport).

- 1 : 1: A mandatory one-to-one relationship where both entities must exist. (Usally, this is not the case in practice, as it would be too restrictive and would require to create both entities at the same time.)

2.4.1.2 One-to-Many (1:N) Relationship

A **one-to-many relationship** occurs when a single instance of an entity relates to multiple instances of another entity. This relationship is commonly used to represent hierarchical or ownership relationships.

For example, consider the relationship between Department and Employee. Each department can have multiple employees, but each employee belongs to a single department. In ORM, the Department entity has a collection of Employee entities, while each Employee contains a foreign key referencing the Department.

The cardinality constraints are the following.

- $1 : N$: A department must have at least one employee.
- $0 : N$: A department can exist without any employees.

2.4.1.3 Many-to-Many (N:N) Relationship

A **many-to-many relationship** arises when multiple instances of an entity are related to multiple instances of another entity. This relationship is often implemented using an intermediate or junction table that connects the two entities, storing the relationships explicitly.

For example, in an academic setting, a Student can enroll in multiple Courses, and each Course can have multiple Students. The Enrollment table acts as the junction table, containing foreign keys referencing both Student and Course entities.

The cardinality for such relationships is:

- $N : N$: Each student can be linked to multiple courses, and each course can be linked to multiple students.

2.4.1.4 Relation Rules and Constraints

To maintain logical consistency and integrity, certain rules and constraints govern the relationships between entities:

1. **Cardinality Rules:** Define the minimum and maximum number of objects allowed on each side of the relationship (e.g., $0 : 1$, $1 : N$, $N : N$).
2. **Ownership Rules:** Specify which entity is the owner or controller of the relationship, determining how changes propagate between related entities.

3. **Cascade Rules:** Define the behavior of related objects when an entity is modified or deleted. For example, cascading deletes ensure that when a Department is deleted, all associated Employee records are also removed.

These constraints are critical in ensuring that relationships remain coherent and reflect the domain requirements. Using ORM frameworks, developers can abstract these rules into high-level representations, reducing the complexity of database interactions [28].

2.4.2 Example of ORM Usage

Script 2.1 is an example of how ORM can be used in Python with Django. It implements the relationships described in Figure 10b.

```
class Person(models.Model):
    id = models.CharField(max_length=255, primary_key=True)

class Teacher(models.Model):
    id = models.CharField(max_length=255, primary_key=True)
    person = models.OneToOneField(Person, related_name='teacher')

class Course(models.Model):
    id = models.CharField(max_length=255, primary_key=True)

class Subject(models.Model):
    id = models.CharField(max_length=255, primary_key=True)
    course = models.ForeignKey(Course, related_name='subjects')

class Assignment(models.Model):
    id = models.CharField(max_length=255, primary_key=True)
    professor = models.ForeignKey(Teacher)
    subject = models.ForeignKey(Subject)

# Adding the Many-to-Many relationship through Assignment
Subject.teachers = models.ManyToManyField(Teacher, through=Assignment, related_name='
    subjects')
```

Script 2.1 – Example of ORM usage in Python with Django.

Both Figure 10b and Script 2.1 show a new object called *Assignment* to manage the relation between *Teachers* and *Subjects*.

Overall, ORM is a powerful tool in modern software development, offering significant benefits in terms of productivity and code maintainability.

2.5 GENETIC ALGORITHMS

Genetic algorithms (GAs) are optimization methods inspired by the principles of natural selection and genetics [30]. These algorithms aim to find solutions to problems by evolving a population of candidate solutions over time. In this section, we will explore the key components of genetic algorithms and the processes involved in evolving solutions.

2.5.1 Genes, Chromosomes, and Population

In genetic algorithms, the concept of a gene, chromosome (or genotype), and population is crucial to understanding how solutions are encoded and evolved [30]. A gene represents the smallest unit of information in a genetic algorithm, similar to how a gene in biology contains hereditary information. Several genes together form a chromosome, which is a complete candidate solution to the problem being addressed. Finally, a population is a set of chromosomes representing a diverse set of possible solutions. Figure 11 visually demonstrates these relationships. Each black rectangle in the diagram corresponds to a gene, and groups of genes form chromosomes. The entire collection of chromosomes is known as the population.

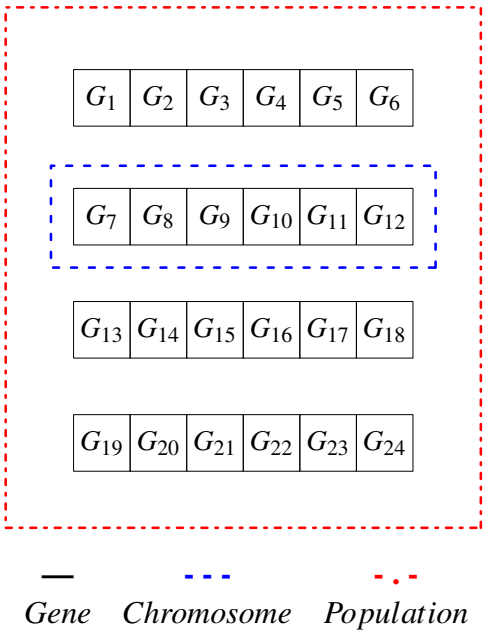


Figure 11 – Representation of all elements used in a genetic algorithm.

In summary, the success of a genetic algorithm heavily depends on how well the problem is encoded into genes, chromosomes, and populations, as these structures form the foundation for all subsequent evolutionary processes.

2.5.2 Mutations and Crossovers

Genetic algorithms rely on two primary genetic operators to evolve the population of chromosomes: mutation and crossover [30]. These operators allow the algorithm to explore the solution space.

The mutation operator introduces random changes to one or more genes within a chromosome. This randomness simulates natural mutations in biological organisms, helping to maintain genetic diversity in the population and explore previously unexplored regions of the solution space.

The crossover operator combines two parent chromosomes to produce offspring. During this process, the genetic material from each parent is exchanged, creating new chromosomes that contain traits from both parents. This operator simulates sexual reproduction in biology, where offspring inherit features from both parents, contributing to the evolution of better solutions over generations.

In conclusion, the mutation and crossover operators are vital for balancing the trade-off between, respectively, exploration (searching new areas of the solution space) and exploitation (refining existing good solutions) in genetic algorithms.

2.5.3 Selection

The selection process in genetic algorithms is responsible for choosing which chromosomes will contribute to the next generation of the population [30]. The selection is guided by an evaluation function, which measures the quality of each solution for the problem, solutions with higher quality are more likely to be selected. This process is crucial for guiding the evolution of solutions toward better fitness levels.

There are several selection methods, including:

- **Roulette Wheel Selection:** Chromosomes are selected based on their fitness proportionally to the total fitness of the population. Higher fitness chromosomes have a higher chance of being selected.
- **Tournament Selection:** A subset of chromosomes is randomly chosen, and the best chromosome from this subset is selected for reproduction.
- **Rank Selection:** Chromosomes are ranked based on their fitness, and selection is performed based on this ranking.

The selection process ensures that better solutions have a higher chance of being passed on to future generations, allowing the algorithm to converge toward optimal or near-optimal solutions over time.

2.5.4 Algorithm

The genetic algorithm is an iterative process that evolves a population of candidate solutions over generations. The algorithm typically follows the steps shown in Figure 12. An inital population is generated, usually by random. Each individual of the population is evaluated by a fitness function, which guides the selection step. The selected individuals are subjected to the genetic operators, and the population is updated with the new individuals. Until a termination criterion is achieved, the steps starting by the evaluation are repeated.

Genetic algorithms are versatile and can be applied to various optimization problems, including scheduling, routing, and function optimization. Their ability to explore large solution spaces and adapt over time makes them powerful tools for solving complex problems.

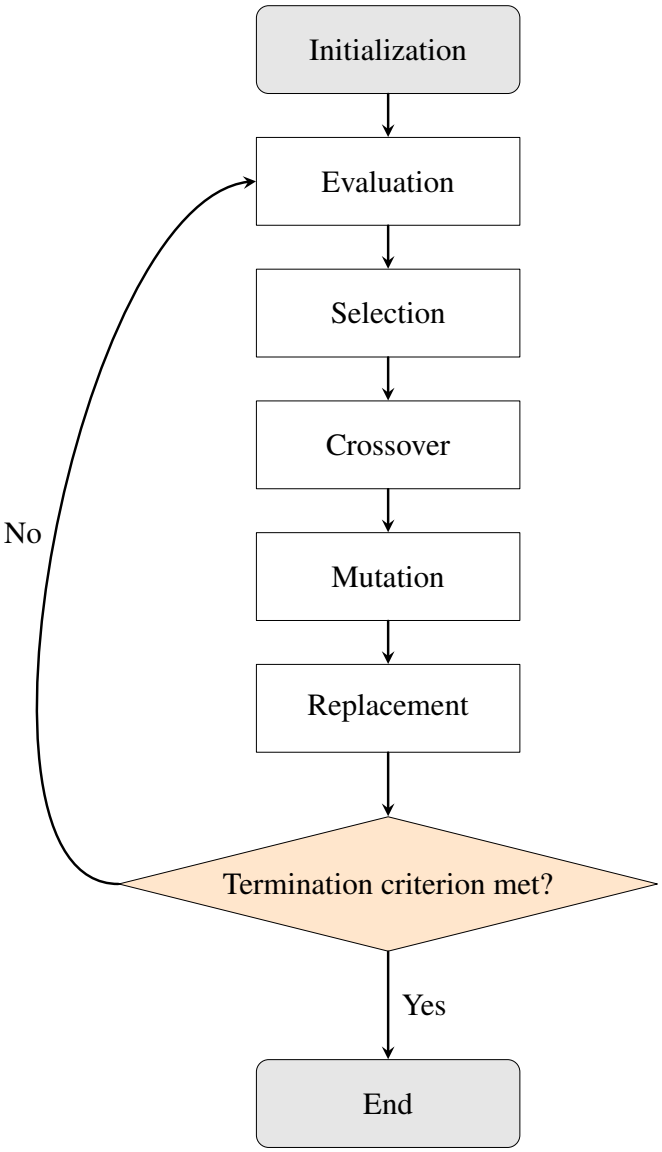


Figure 12 – Flowchart of a Genetic Algorithm.

3 CHARACTERIZATION OF GROUPING PROBLEMS

In this chapter, we explore the characterization of grouping problems, focusing on their definitions, common types, and recent developments. We begin by defining how grouping problems are approached in industry. This concept is then related to graph theory, where vertices represent objects and edges represent connections, providing a theoretical framework for understanding grouping problems [31].

We then delve into common types of grouping problems, including explicit matching, implicit matching, one-to-one matching, one-to-many matching, many-to-many matching, user-item matching, and others [32, 33]. Each type presents unique challenges and applications in various industries.

Finally, we discuss recent developments in the field of grouping problems, including advanced algorithms for solving matching problems efficiently and applications in diverse fields such as quantum computing [34], artificial inteligence [35, 36], and Humanitarian and Resource Allocation [37]. By examining these developments, we aim to provide a comprehensive overview of the current state of grouping problems and their significance in various domains [1].

3.1 HOW TO DEFINE AN INDUSTRY MATCHING (GROUPING) PROBLEM

In industry, matching problems are problems where the objective is to match some elements, usually the optimizing an objective function. Importantly, in industry, the concept of matching problems is often not bounded to the graph theory definition of matching problems. For that reason, we refer to grouping problems as a larger set of problems, which includes not only graph theory's matching problems, but also other problems that can be represented as a graph, where the objective is to group objects based on some criteria.

When faced with a set of objects, such as job openings and candidates, the objective is to group these objects, assigning candidates to jobs in this context, where each group generates a set of statistics, and this statistics are used to optimize an objective function. This concept is well-established in the field of operations research and combinatorial optimization [38].

3.1.1 Job Assignment

One of the most well-known examples of a grouping problem is job assignment. The objective is to match job positions with candidates who possess the necessary skills, experience, and qualifications. The process involves considering the specific requirements of each position and the attributes of the candidates, aiming to maximize overall efficiency or satisfaction. This type of problem has been widely studied, particularly in the context of labor markets and automated recruitment systems [39].

3.1.2 Resource Allocation

Resource allocation problems involve distributing limited resources—such as rooms, equipment, or funds—among competing tasks or projects. The challenge lies in ensuring that the allocation maximizes efficiency while meeting constraints, such as availability and demand. These problems are critical in industries like healthcare, manufacturing, and project management, where effective resource distribution directly impacts outcomes [40].

3.1.3 Network Routing

In network routing problems, the goal is to determine the optimal way to route data, goods, or resources through a network. This involves minimizing costs, delays, or energy consumption while ensuring that demands are met across the network. Such problems are particularly relevant in logistics, telecommunications, and supply chain management, where optimizing flow is essential for performance [11].

3.1.4 Stable Matching

Stable matching problems focus on creating pairings where no two entities would prefer being matched with each other over their current assignments. The stable marriage problem is a classic example, where the objective is to create stable pairs based on mutual preferences. Applications of stable matching range from college admissions processes to organ donation programs, where stability and fairness are crucial [32, 33]. These problems are fundamental in various industries and have been extensively studied [38].

3.1.5 Industrial Terminology compared to Graph Theory

In the context of graph theory, vertices may represent objects, and edges represent the connections or relationships between these objects [5, 6]. This basic terminology is crucial when translating industrial problems into graph models.

Edges often contain statistics that can represent preferences, costs, capacities, and other characteristics. For example, in the stable marriage problem (see Subsection 3.1.4), edges' statistics represent preferences between pairs [41]. In the Hungarian algorithm for the assignment problem (see Subsection 3.1.2), edges represent costs associated with assignments [42]. In network flow problems (see Subsection 3.1.3), edges represent capacities that limit the amount of flow through the network [10].

For industry applications, any graph cover problems are often collectively referred to as matching problems [28, 27]. However, to avoid confusion, in the context of this work, we will use the terms “grouping” and “matching"" interchangeably.

For instance, in a logistics network, vertices could represent warehouses and stores, while edges represent possible delivery routes. Finding an optimal set of routes that minimizes cost can be modeled as a grouping problem. Using this terminology allows us to frame complex

industrial problems in a structured way, making it easier to apply graph-theoretical algorithms and techniques to find optimal solutions.

3.2 USUAL MATCHING PROBLEMS

In this section, grouping problems will be referred to as "matching" problems, as this is how they are often called in the industry. The categorization of problems into subcategories, based on the classification provided by the survey [1], reveals several common grouping problem structures, such as explicit and implicit matching. Table 1 shows a detailed relation of problems, characteristics, and applications. Other characteristics that can be applied to all problems include online/offline (dynamism), scalability, heterogeneity, noise tolerance, and privacy requirements [1].

Category	Sub-category	Algorithms	References	Applications
Explicit Matching	One-to-one Matching	-	[41, 43, 44, 45]	marriage market
	Many-to-one Matching	-	[45, 41, 43, 46, 47]	job matching
			[48, 49, 50]	pay matching
	Many-to-many Matching	Optimization algorithms	[49, 45, 43, 44]	cognitive radio networks; D2D communications
Implicit Matching	Retrieval Matching	Traditional matching algorithms	[51, 52, 44, 53, 54, 45]	machine translation; expertise matching; question-answer matching
		Representation-based algorithms	[55, 55, 56, 57]	
		Interaction-based algorithms	[53, 55, 57, 57, 44], [49, 57, 53, 43, 44]	
	User-item Matching	Basic algorithms	[58, 57]	recommendation systems
		Representation-based algorithms	[59, 53, 60, 53, 43]	
		Matching function-based algorithms	[61, 55, 62, 49]	
	Entity-relation Matching	Factorization-based algorithms	[53, 63, 49, 49]	recommendation systems; knowledge fusion; information retrieval
		Neural network-based algorithms	[46, 61, 44, 43]	
		Translational distance-based algorithms	[46, 43, 53, 45, 43]	
	Image Matching	Area-based algorithms	[61, 64, 53, 53, 65]	robot vision; object recognition; medical image diagnosis
		Feature-based algorithms	[61, 66, 49, 44, 46], [46, 43, 61, 43, 43]	

Table 1 – Summarisation of Matching Algorithms and Applications According to [1].

Matching problems can be broadly divided into two main categories: explicit and implicit.

3.2.1 Explicit Matching

Explicit matching problems involve objects that have preferences about whom they prefer to group with [1, 38]. These can be further categorized into:

- **One-to-one**: also known as bipartite matching, common in job assignment where each job is assigned to a single candidate and vice versa [42].
- **Many-to-one**: found in scenarios like college admissions, where multiple students can be assigned to a single college [41].
- **Many-to-many**: occurs in contexts such as organ donation, where multiple donors can provide organs to multiple recipients [67].

3.2.2 Implicit Matching

Implicit matching focuses on calculating the 'score' of the grouping without explicit preferences, optimizing based on a grouping function [1]. Examples include:

- **Retrieval Matching**: in information retrieval, it involves users inputting queries that express their needs and obtaining the desired information from a search engine's database [68].
- **User-item Matching**: in recommender systems, it helps users obtain items of interest accurately [69].
- **Entity-relation Matching**: it involves the use of knowledge graphs in applications such as semantic parsing, information extraction, link prediction, recommender systems, and question answering [70].
- **Image Matching**: it compares different images to identify similarities or correspondences, used in fields like computer vision and pattern recognition [71].

3.3 RECENT DEVELOPMENTS

The field of matching problems is constantly evolving, with ongoing research introducing new problems, solutions, and algorithmic approaches to address the complexity and diversity of these challenges in industry. In this section, we outline a preliminary review aimed at identifying the latest developments in matching problems, focusing on advancements made since 2021.

To capture the most recent and relevant studies, we conducted an initial search in Google Scholar on January 19, 2025, using the following search string:

> (TITLE-ABS-KEY("pairing") OR TITLE-ABS-KEY("matching") OR TITLE-ABS-KEY("grouping")) AND (TITLE-ABS-KEY("solver") OR TITLE-ABS-KEY("algorithm")) AND TITLE-ABS-KEY("graph") AND PUBYEAR AFTER 2021

The query was executed using a command-line Python tool named `scholar.py`[1], which automates the retrieval of publication metadata such as titles, abstracts, and citation counts. This approach enabled a structured and replicable filtering process.

The review process involved applying inclusion criteria to select studies that contribute to the development of matching algorithms, whether through incremental improvements, novel problem types, algorithmic frameworks, or solvers that expand the applicability of matching techniques in industry.

After collecting the top 100 studies by number of citations, they were filtered by language, retaining only those in English. Then, based on titles and abstracts, works unrelated to matching

[1] `scholar.py` is an open-source command-line utility that allows querying Google Scholar and retrieving citation metadata programmatically. See: <https://github.com/ckreibich/scholar.py>

were removed, including six that focused on diverse clustering problems. Although clustering problems may be addressed by the general solution proposed in Chapter 4, they are out of scope and suggested as future work in Chapter 7. Figure 13 illustrates the selection process, showing the number of studies removed and retained at each stage. Furthermore, we analyze the most prominent topics that emerged from the remaining studies.

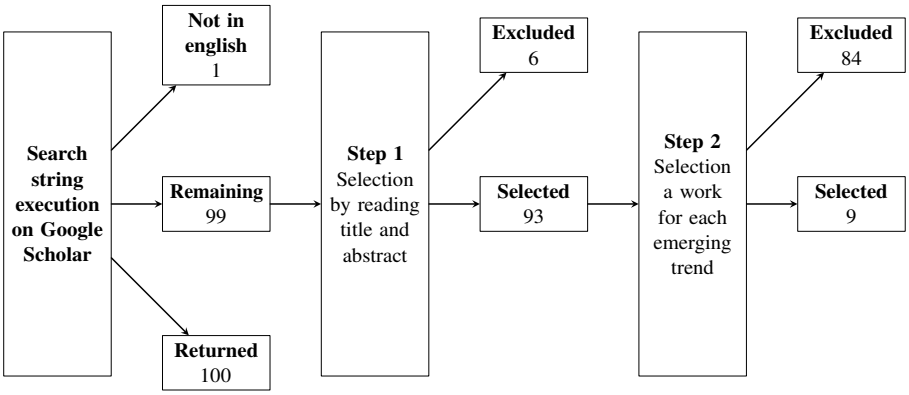


Figure 13 – Selection Process of the Articles.

3.3.1 Overview of Recent Existing Approaches

Table 2 presents a summary of the selected studies in step 1, categorizing them based on their focus and contributions to the field of matching. Each study is classified according to the definitions of Implicit Matching by [1]. Additionally, we detail the common subjects treated by the new publications.

Implicit Matching Category	Total
Retrieval Matching	16
User-item Matching	10
Image Matching	33
Entity-relation Matching	34

Table 2 – Total of Works after step 1 by Implicit Matching Category.

This review aims to demonstrate the breadth of recent developments in matching, highlighting both the versatility of applications and the growing need for centralized tools that simplify the implementation and adaptation of matching solutions across various industrial contexts.

3.3.2 Emerging Trends in Graph Matching Algorithms

Here we overview the 9 selected studies about emerging trends.

1. **Quantum Computing**

- **Trend:** Quantum computing is starting to impact optimization and matching algorithms. Some works suggest that quantum algorithms are being explored for graph-based problems, particularly in sparse data and matching contexts [34].
- **Context:** Quantum algorithms may offer exponential speedup over classical methods in certain areas, and combining quantum techniques with combinatorial optimization is a growing field.

2. **AI and Deep Learning in Graph Matching**

   - **Trend:** Many studies focus on graph neural networks (GNNs) and deep learning in graph matching [35, 36].
   - **Context:** AI, particularly deep learning, is being applied to improve the accuracy and efficiency of graph matching, especially in image recognition, NLP, and multi-modal data integration.

3. **Video and Image Recognition**

   - **Trend:** Recognition algorithms using graph matching are becoming more common, especially in computer vision [72].
   - **Context:** This is closely related to AI, where graph-based methods are used for tasks like multi-object tracking and 3D object detection.

4. **Combinatorial Algorithms and Optimization**

   - **Trend:** Classical combinatorial algorithms remain an important area of research [73].
   - **Context:** These algorithms are central in problems like scheduling, resource allocation, and network flow.

5. **Networks and Multi-Agent Systems**

   - **Trend:** Network theory and multi-agent systems use graph matching for optimization [74, 75].
   - **Context:** These topics focus on applications in wireless communication, sensor networks, and privacy-preserving computation.

6. **Cross-modal and Multi-modal Matching**

   - **Trend:** Matching across different types of data (e.g., text and images) is an emerging trend in AI and cross-modal retrieval [36].
   - **Context:** These algorithms are applicable in industries like e-commerce, digital media, and autonomous systems.

7. **Humanitarian and Resource Allocation Applications**

    - **Trend:** Graph matching and optimization algorithms are used in resource allocation and humanitarian problems [37].
    - **Context:** This trend focuses on real-world applications like disaster management and healthcare.

8. **Graph Matching for Knowledge Graphs and Semantic Systems**

    - **Trend:** Knowledge graphs and semantic matching are gaining attention in AI reasoning and data integration [76].
    - **Context:** These methods are key in NLP, semantic web technologies, and AI-driven data systems.

9. **Privacy and Security in Graph Matching**

    - **Trend:** Privacy-preserving graph matching is a significant concern, particularly for sensitive data [75].
    - **Context:** These methods are relevant in domains like healthcare, finance, and social networks.

4 DESIGNING A PIPELINE FOR SOLVING GROUPING PROBLEMS

Grouping problems are a common challenge in various fields, requiring an effective and efficient approach for their resolution. To achieve this, it is essential to represent these problems in both a simple and generic manner. Simplicity ensures ease of understanding, while a generic approach guarantees that the representation can encompass the majority of grouping problems encountered.

In this chapter, we propose using the SCF (Set Coverage Framework), which uses the Set Cover problem (see Section 2.3) as the foundational representation for all grouping problems. The optimization version of the Set Cover problem is NP-Hard, indicating that any decidable grouping problem can be reduced to it [23]. This approach provides a unified framework that simplifies the conceptualization and handling of diverse grouping problems.

To manage the specific properties of each problem, we introduce statistics that can identify the nuances of the problem, such as the number of groups, whether the problem is weight-based or capacity-based, whether an object is unique or can be in multiple groups, and other relevant attributes. These statistics allow for the customization and fine-tuning of the problem representation, facilitating the reduction of complex problems to simpler forms, potentially even to P problems, enabling the use of specialized solvers [1].

Furthermore, we discuss the application of a metaheuristic solver for the SCF problem, capable of handling all cases, including those where problem reduction is not feasible [77]. This solver ensures a robust and versatile solution approach, even for the most challenging instances.

This chapter aims to lay the groundwork for a systematic pipeline in solving grouping problems, offering a comprehensive representation and a flexible, powerful solution mechanism.

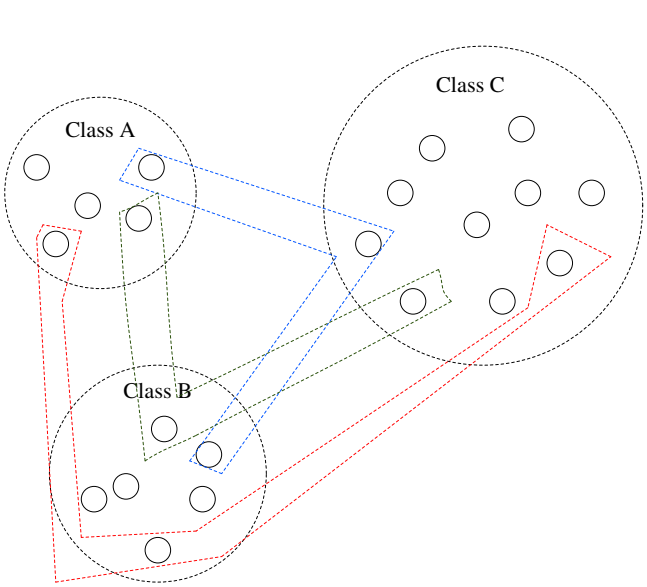
4.1 GROUPING PROBLEMS AS THE SET COVER FRAMEWORK

The Set Cover Framework is a way to represent grouping problems as the the Set Cover problem (see Section 2.3) and its variants thorough objects and its relations, since they are straightforward to be represented in this manner as shown in Figures 14a and 14b. The weighted version of the Set Cover can consider the weights as being defined and computed based on the relation of those objects.

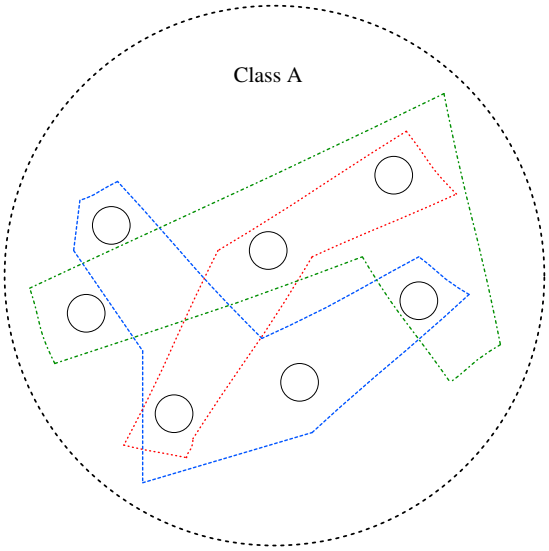
In addition, every decidable grouping problem can be seen as a set cover problem. While it sounds as a bold statement, this is in fact, a trivial one, due to the fact that the decision version of the Set-cover is a NP-Complete problem [2]. It implicates that any decidable grouping problem can theoretically be reduced to it [23].

4.1.1 Representation

As the main purpose of Grouping Problems is to group objects, they can intuitively be represent as a relational diagram. Figure 15 shows how to represent several grouping problems



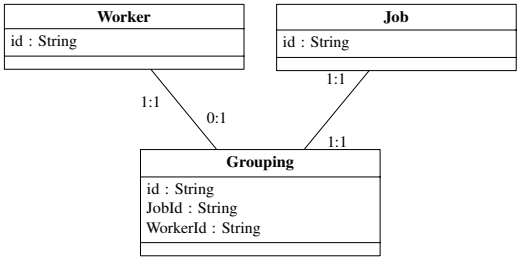
(a) Set Coverage Over Elements from Multiple Classes.



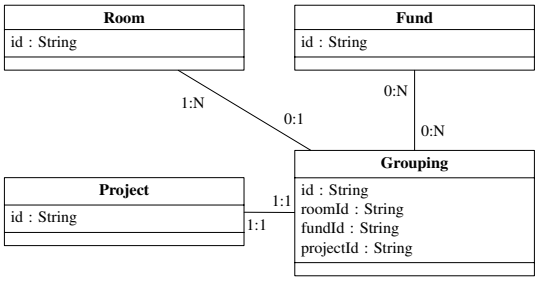
(b) Set Coverage over Elements from a Single Class.

Figure 14 – Visualizing Set Cover Problems in the context of Classes and its relations. Every small circle represent an instance of a class. The colored boxes represent both relations between classes and sets.

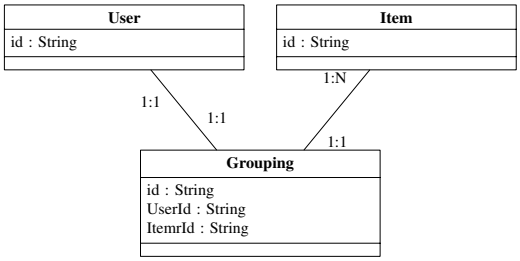
earlier discussed in Chapter 3. Note that the grouping problems can always be represented as a many-to-many relation. Here we do not refer to the many-to-many matching from Chapter 3, but we refer to the many-to-many object relation, explained at Section 2.4.



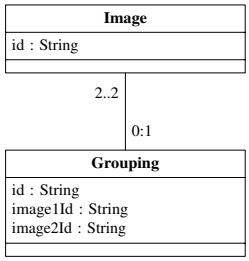
(a) Job Assignment Problem.



(b) Resource Allocation Problem.



(c) User-Item recommendation Problem.

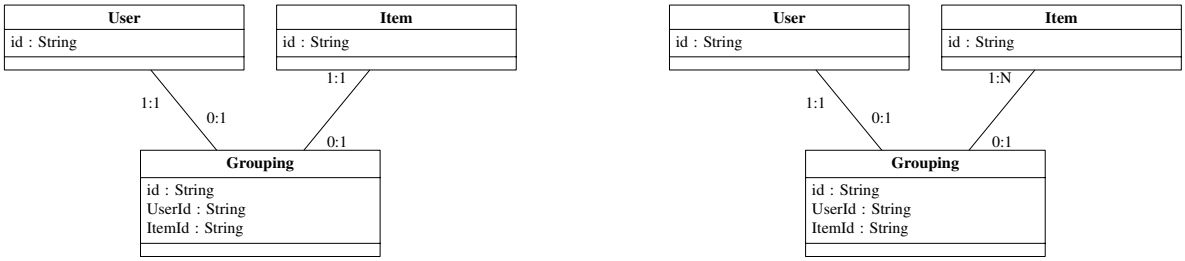


(d) Image Matching Problem.

Figure 15 – Visualizing Grouping Problems as a relational diagram.

It is important to also properly define the geometry (dimension) of the relations, as they can represent different problems, as shown in Figure 16.

As we discussed before, sets can be collections of instances, which do not necessarily



(a) An user has to be assigned to a single item. (b) An user can be assigned to multiple items.

Figure 16 – The User-Item Recommendation Problem showcases how different dimensions on the relations result in different restrictions. In both examples a grouping must have an user, but not all users have an assignment.

belong to the same class, which represent relationships among these instances. Using this parallel, it is possible to see a Set Cover Problem and its variants as a relational diagram too, for instance, Figure 17 is the relational diagram version of the set cover problem shown at Figure 14a, while Figure 18 is the relational diagram version of the set cover problem shown at Figure 14b.

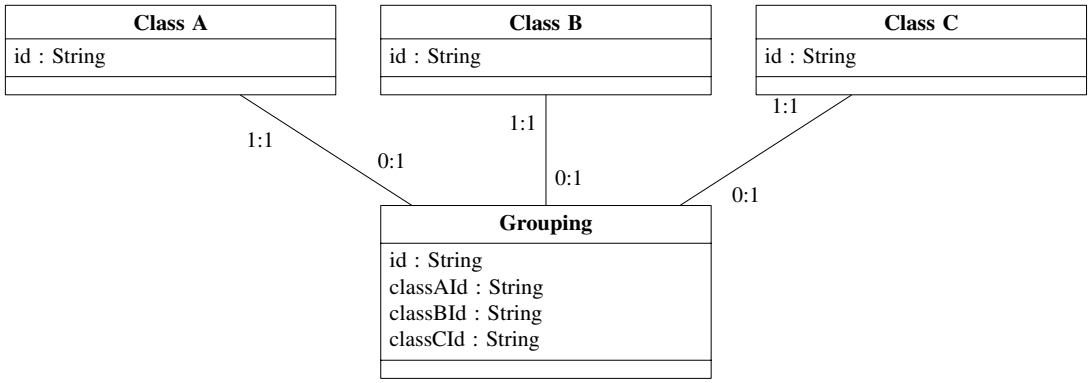


Figure 17 – Representation of the relation between 3 differents Classes.

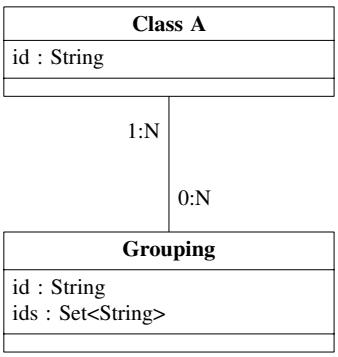


Figure 18 – Representation of the relation between elements of a same Class.

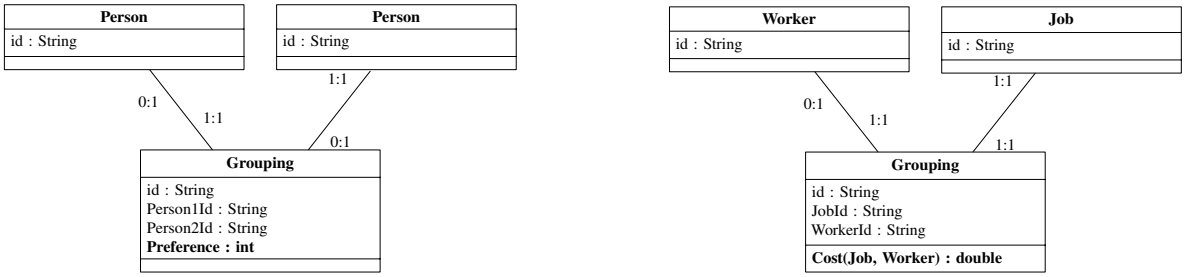
To summarize, the SCF is the way to represent the grouping problems as relation between objects, which is also the same way one can represent a set cover problem. It permits the representation of every decidable grouping problem [23] and also makes it more practical to understand and use it, since the object relation paradigm is already well established in the industrial environment.

The solution for a grouping problem in the SCF can be represented as a list of grouping objects.

4.1.2 Statistics

Depending on the problem, each grouping can encapsulate various statistics such as costs, capacities, number of elements, qualities, and validity, among other characteristics relevant to the grouping problem. At the SCF the statistics are attributes of the *Grouping* class.

It is crucial for these statistics to be well-defined beforehand for a specific problem. Depending on the nature of the problem, these statistics can either be computed on-demand or predefined, depending upon whether preferences are implicit or explicit. Figure 19 shows how this statistics can be represented.



(a) Stable Marriage Problem. (b) Job Assignment Problem.

Figure 19 – Implementing Statistics at the Grouping Class.
The statistics in each grouping class are bolded.

At Figure 19a, showing the Stable Marriage Problem, a grouping represents potential relationships between pairs, where the statistics of these groupings reflect preferences. In the context of the Job Assignment Problem, at Figure 19b, a grouping represents potential assignments between workers and jobs, with the statistics indicating the costs associated with each assignment.

4.1.3 Objective Function

In optimization problems, the objective function is a mathematical expression that quantifies what needs to be optimized. The goal of an optimization problem is to find the values (variables) that either maximize (fitness function) or minimize (cost function) the result of this function, subject to certain constraints.

Formally, if x represents a vector of decision variables, the objective function $f(x)$ is a function mapping the decision variables x to a real number. The optimization process involves finding the values of x that either maximize or minimize $f(x)$, depending on the problem.

In the case of this work, the objective function is computed based on the statistics of every grouping used in a possible solution for a given problem. Lets assume a possible solution is a list of groupings L. An objective function could be like the Algorithm 1.

Algorithm 1 Example of Objective Function - Maximize the Return

```
1: Sum ⇐ 0
2: for each Grouping in L do
3:     Sum ⇐ Sum + value of Grouping
4: end for
5: return Sum
```

Algorithms 2 and 3 show a possible objective function for the job assignment and stable marriage problems, respectively. They have very different behavior, while Algorithm 2 focus on finding low cost assignments and wants to minimizes the function, Algorithm 3 focus to find the biggest amount of stable matchings through maximizing the function.

Algorithm 2 Job Assignment Objective Function - Minimize the Return

```
1: if amount of Assignments ≠ amount of Jobs then
2:     return ∞
3: end if
4: Sum ⇐ 0
5: for each Assignment in Assignments do
6:     if amount of Workers in Assignment ≠ 1 or amount of Jobs in Assignment ≠ 1 then
7:         return ∞
8:     end if
9:     Sum ⇐ Sum + cost of Assignment
10: end for
11: return Sum
```

Note in Algorithm 2 the return of infinity in lines 2 and 7 enforce hard constraints, respectively, all jobs must be assigned (lines 1-3) and each assignment must have a single worker and single job (lines 6-8). Soft constraints could be implemented by increasing the value of sum with a penalty values instead of returning infinity.

Similarly, Algorithm 3 enforces hard constraints by returning zero (maximum function) in lines 3, 8 and 16.

4.1.4 Problem Specification and Configurations

As a lot of grouping problems have similar characteristics as shown in Chapter 3. Due to this observation, it is possible to create template statistics with a default behavior. Such as:

- **Cost:** it defines the cost to make a grouping. Usually the objective is to minimize the sum of the costs, while having the bigger amount of grouping (priority one).
- **Preference:** it is based on preferences of objects. Have the greater amount of groupings while respecting preferences of objects, where none objet wants to switch to a different grouping.
- **Usage Limit:** it defines how many different groupings an element can be part of.

Algorithm 3 Stable Mariage Objective Function - Maximize the Return

```
1: for each Marriage in Marriages do
2:     if amount of People in Marriage ≠ 2 then
3:         return 0
4:     end if
5: end for
6: for each Person in Marriages do
7:     if Person appears more than 2 times then
8:         return 0
9:     end if
10: end for
11: for each Person_1 in Marriages do
12:     PairPerson_1 ⇐ pair of Person_1
13:     for each Person_2 in Marriages do
14:         PairPerson_2 ⇐ pair of Person_2
15:         if Person_1 prefers Person_2 over PairPerson_1 and Person_2 prefers Person_1 over
            PairPerson_2 then
16:             return 0
17:         end if
18:     end for
19: end for
20: return amount of Marriages
```

- **Ranking:** it defines how similar objects are, maximize the average ranking.

4.2 PIPELINE

Translating grouping problems to the SCF (Set-Cover Framework) is not sufficient; it is also crucial to solve the underlying optimization problems. However, each grouping problem presents its own characteristics, which in turn can be better tackled with a particular solver. Furthermore, for different solvers, different internal representations of the problem are required. To address these problems, a specialized pipeline is employed. Figure 20 illustrates a pipeline that demonstrates how a grouping problem, represented using the SCF framework, can be assigned to a specialist solver and then returned to the same SCF representation. First the SFC is sent to the assigner, that will choose a solver based on the statistics of the problem, each solver is an specialized algorithm with a converter to convert the SCF to the expected input format to the algorithm, after that a normalizer will get the output of the algorithm and translate it again to the SCF, outputting a list of groupings objects.

4.2.1 Assigner

Since the problems are represented using the SCF, one might assume that only NP-hard solutions are available. However, due to the information stored in the statistics, it is sometimes possible to reduce the problem to a simpler one.

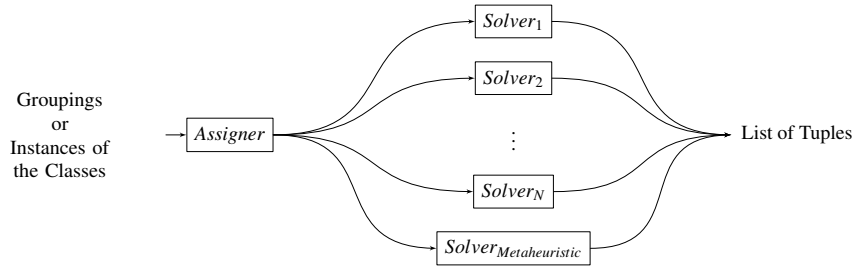


Figure 20 – A pipeline to solve Grouping Problems.

The assigner gathers information from the statistics and metadata from the grouping to identify a specific and specialized solver for that particular grouping problem, if no one is available, a metaheuristic one is assigned.

Figure 21 shows a simplified version of the assigner; a more advanced assigner can be implemented in future work. For this example, the assigner analyzes the size of the grouping, if the relations have preferences (i.e. each element has a preference for which element to be grouped with), if the relations have costs (i.e. each element has a cost to be grouped with another element), and if there is a ranking (i.e. each element has a ranking with respect to the other elements). Based on these characteristics, it selects the appropriate solver.

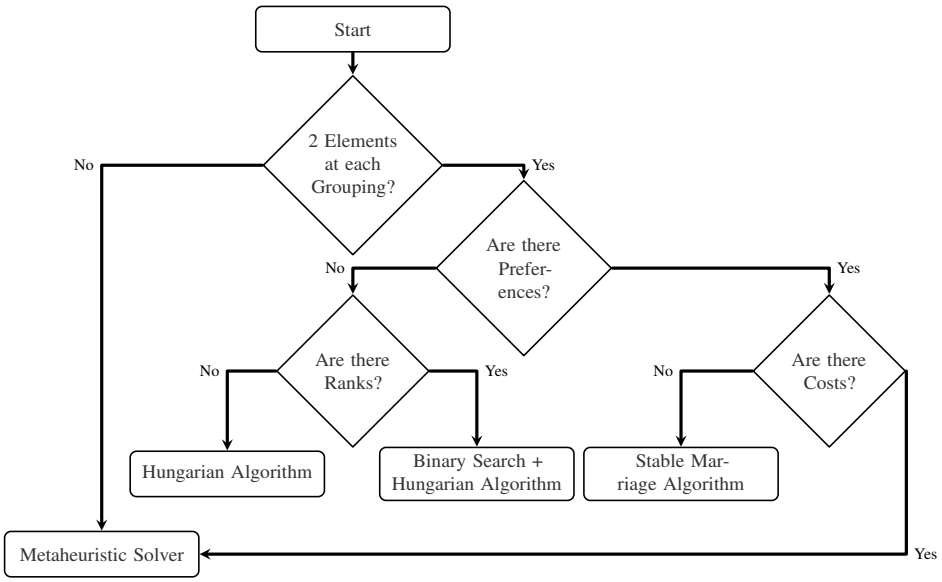


Figure 21 – Example of a possible flowchart for the Assigner. The solvers shown here are explained at Section 6.2.3.

4.2.2 Solvers

According to the problem identified by the assigner, it is important to have algorithm-specific solvers. In cases where there is no predefined specification for the problem, a metaheuristic algorithm based on genetic algorithms will be used to solve the problem.

Algorithm-specific solvers are tailored to solve particular types of problems more efficiently than general-purpose solvers. One such solver is the Hungarian algorithm, which is

used for finding the optimal assignment in a bipartite graph [42]. This algorithm ensures that the total cost of the assignment is minimized, making it particularly effective for solving assignment problems where each element in one set must be matched with an element in another set at minimal cost.

For cases where the objective is to minimize the maximum difference among the matchings, a combination of binary search and the Hungarian algorithm is employed [78]. The binary search is used to find the optimal threshold, and for each threshold value, the Hungarian algorithm determines if a valid matching exists within that threshold. This combined approach allows for efficient minimization of the maximum difference among the matchings.

Another important algorithm-specific solver is the Stable Matching Algorithm, also known as the Stable Marriage algorithm [41]. This algorithm is used for finding stable matchings, where there are no two elements that would both prefer each other over their current matches. The Stable Marriage algorithm is widely applied in scenarios such as matching students to schools or residents to hospitals, ensuring that the matchings are stable and no participants have an incentive to deviate from their assigned matches.

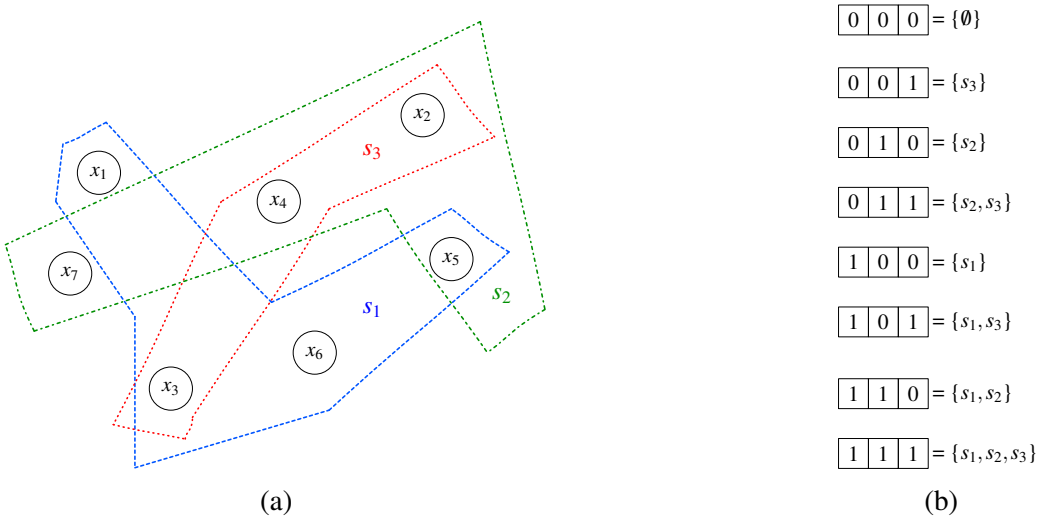
4.2.3 Metaheuristic Solver: Genetic Algorithm

In situations where no specialized solver is available, a metaheuristic approach, such as a genetic algorithm, is used. This method attempts to find a near-optimal solution by mimicking the process of natural selection, iteratively improving the solution through processes such as selection, crossover, and mutation [30]. Section 2.5 gives a better understanding on how the genetic algorithm works.

4.2.3.1 Gene and Chromosome Representation

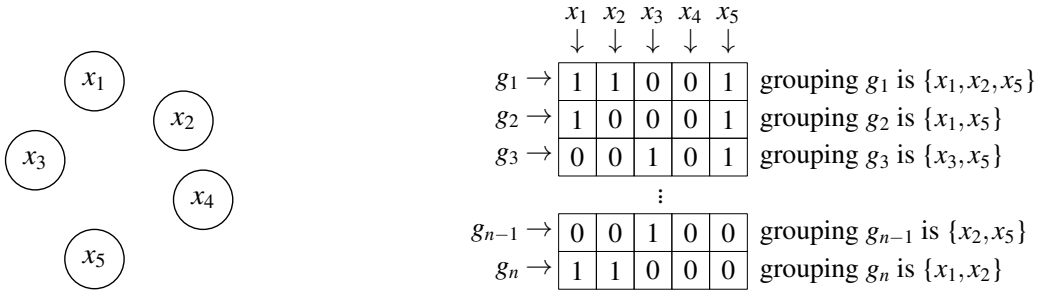
The design allows the user to enter two types of input, the allowed Grouping Objects, or all instances of classes that can be part of the groupings. For the first case, the chromosome is a bit mask b of size n, where n is the number of allowed Grouping Objects. The i-th bit in b indicates whether or not the i-th grouping object is selected at that solution option. Figure 22 shows an example of it. For the second case, the chromosome is a list of n bit masks (a matrix). Therefore, we have a predefined maximum of n grouping. Each bit mask, of size m, represents a grouping, where m is the number of instances available. The i-th bit of the j-th bit mask indicates whether the i-th instance is part of the j-th grouping. Therefore, the chromosome is a matrix of size $n \times m$. This can generate invalid groupings, so they need to be checked after mutations and cross-overs. Figure 23 shows an example of it.

These representations are not optimized, but can cover all the cases. As these representations are binary, there is no need to elaborate specific mutations and crossover operations, since the literature already implements many operations (e.g. single-point crossover, multi-point crossover, flip mutation) [79].



(a) (b)

Figure 22 – Example of a chromosome from a problem where the allowed Grouping Objects are predefined. (a) Allowed Grouping Objects are s_1, s_2, and s_3. (b) Several chromosomes binary representation. Each line is a different chromosome that represents the selected groups.



(a) (b)

Figure 23 – Example of a chromosome when there are no allowed grouping objects predefined. (a) Instances of arbitrary classes. They are x_1, x_2, x_3, x_4, and x_5. There are no predefined allowed grouping objects. (b) A single chromosome represented as a binary matrix for $m = 5$. It represents which instance is part of each grouping.

4.2.4 Mapping Inputs and Outputs of Solvers

Each *Solver* needs to translate the input from the SCF to something that its optimization algorithm can comprehend and optimize. It adapts the SCF to the particular requirements of the algorithm. This step is essential for making the data usable by the algorithm in question, as different algorithms may have unique input formats or requirements.

On the other hand, there is also need to translate the output to a SCF, ensuring consistency and compatibility with the SCF representation. It outputs a list of groupings, which optimizes the grouping problem. This final step ensures that the results are presented in a organized manner, facilitating further applications.

5 FAIR MATCHING

Matching in bipartite graphs refers to determining optimal combinations between elements of two sets, U and V, often considering predefined constraints and specific preferences. This paradigm is frequently applied in contexts such as resource allocation, network design, and matching in distributed systems, which demonstrates its versatility by offering solutions to a wide range of practical problems [2].

One of the conventional approaches to solving bipartite matching relies on reducing the problem to a Minimum Cost Max Flow (MCMF) problem, where several algorithms may be employed. However, it does not often consider attributes among specific sub-sets of U and V, which may possess distinct traits that are essential in various contexts. Considering such sub-sets' characteristics makes the bipartite matching problem relevant to a large set of real-world applications, we call this the *Fair Bipartite Matching Problem*.

We focus on the *minimum quota* characteristic among sub-sets. The *minimum quota* defines the minimal amount of elements of each sub-set that must be matched. By incorporating *minimum quotas* for sub-sets, the proposed method aims for both: operational efficiency and equity in the distribution of resources among sub-sets. In this chapter, we present a reduction from the Fair Bipartite Matching Problem with *minimum quota* to the MCMF.

The practicality of this approach is evident in various scenarios where both qualitative and quantitative criteria are crucial in the pairing process. A prominent example is in recruitment, where evaluating a potential employee's soft skills is as important as assessing their hard skills through objective methods like exams. Another clear application is in network optimization, particularly in selecting the best data centers for different applications. Factors such as backup type and location are just as important as latency in making these decisions.

Several works already address the concept of fairness, which has become increasingly important. However, most approaches rely on heuristic [80, 81, 82] or metaheuristic methods [83, 84, 85]. Here, however, we develop and present a deterministic model in polynomial time, introducing an innovative approach for solving the Fair Bipartite Matching problem. The central proposal involves applying the concept of Minimum-Cost Maximum Flow (MCMF) through an effective mapping of the problem, allowing for the incorporation of fairness criteria, such as minimum quotas for specific sets.

5.1 RELATED WORKS

Several methods have been proposed to address problems of matching and fair allocation. Table 3 summarizes some of the most relevant approaches in terms of accuracy, fairness, and the use of minimum quotas.

The methods proposed by Ahuja et al. (1993) [11], Edmonds and Karp (1972) [2], and Tarjan (1997) [13] focus primarily on matching accuracy without considering fairness or specific quotas.

Methods	Exact	Fair	Minimum Quotas
Proposed	**Yes**	**Yes**	**Yes**
[11]; [12]; [13]	Yes	No	No
[80]; [86]; [87]; [83]; [88]; [89]	No	Yes	No
[90]	No	Yes	Yes
[91]	Yes	Yes	No

Table 3 – Comparison of Allocation Methods, proposed and from literature, in Terms of Accuracy, Fairness, and Minimum Quotas.

On the other hand, Hatfield and Milgrom (2005) [80], Rostami et al. (2018) [86], Pappalardo and Conitzer (2020) [87], Manlove (2017) [83], Yan et al. (2021) [88], and Chapman et al. (2017) [89] address fairness in matching but do not include specific quotas in their formulations.

Sankar et al. (2021) [90] present a method that considers both fairness and minimum quotas, but without exact matching. García-Soriano and Bonchi (2020) [91] propose an exact and fair method that does not incorporate quotas.

Our method proposes an approach that is exact, fair, and respects minimum quotas, promoting efficient and equitable allocation, inspired by the fairness set concept as proposed by Sankar et al. (2021) [90].

5.2 SOLUTION MODELING

As discussed in Section 2.2.4, the MCMF algorithm is traditionally employed to solve matching problems without considering the concept of fairness. We present how to extend the use of MCMF to the proposed concept of fairness. This extension will be achieved through a mapping that constructs a graph to be processed by an MCMF algorithm to solve the original problem.

5.2.1 Objective

Based on the intrinsic characteristics of the MCMF, this method aims to maximize the flow, and among all maximum flows, select the one with the minimum cost. In our case, we search for maximum flow to ensure fair selection in the matching, with the guarantee of respecting the predefined minimum quotas. In situations where there are multiple ways to achieve this allocation, our aim is to minimize the associated matching costs.

The solution aims to meet the following points:

1. **Comply for Minimum Quotas**: Ensure that the predefined minimum quotas for all subsets are fully fulfill (if possible) during the matching process, promoting equity and inclusion.

2. **Cost Minimization**: In situations where there are various matching alternatives, the objective is to minimize the costs associated with these matchings, providing operational

and economic efficiency.

5.3 MODIFICATIONS TO INCLUDE FAIRNESS IN BIPARTITE MATCHING

The fundamental changes made to the mapping aim to make it impossible for the flow to avoid paths with minimum quota elements (in other words, to force the flow through paths with minimum quota elements), as well as to prevent the repeated selection of the same element. Subsequently, the reduction of the Fair Matching problem to MCMF is addressed, along with the proofs of the modeling.

5.3.1 Minimum Quotas

The sets of minimum quota members are defined as subsets of vertices of the same class (e.g., workers) that share one or more characteristics (e.g., workers with disability) and have a minimum number of *matches* if the maximum match is achieved.

Each set of minimum quota members is represented by a vertex, where the incoming edge has a capacity equivalent to the number of *matches* the set needs to obtain, with an associated cost of zero. This configuration encourages the passage of the maximum flow through this vertex, as part of the total flow is required to transit through the quota vertices. Considering a graph with a total possible of N *matches* and having $|Q|$ sets of quotas, where the i-th quota, q_i has n_{q_i} slots. In Equation 1 we define the total matches assigned to quota slots, N_Q as:

$$N_Q = \sum_{i=1}^{|Q|} n_{q_i} \leq N \tag{1}$$

Thus, a set with n_{q_i} quota, with the described mapping, ensures that the other vertices can only contribute a maximum of $N - N_Q$ flow units. This guarantees that if there is a maximum flow passing through the quota set, it will be selected. In Figures 24a and 24b, these vertices are shown in purple / loosely dash-dotted.

Here, N represents the total number of possible matches, typically defined by the smaller of the two sets (demand or supply). In our case, demand (V) is always less than or equal to supply (U), so N is at most the size of the demand set V. However, depending on the graph's structure and quota requirements, it is possible that not all demand is matched, meaning N can be equal to or less than V.

5.3.2 Wide Competition (WC)

The Wide Competition (WC) indicates the number of *matches* that have not been previously assigned to any quota set. Let N be the total possible matches overall, and N_q the total

matching assigned to quota slots. The wide competition slots, n_{wc}, is defined by Equation 2.

$$n_{wc} = \begin{cases} N - N_Q & \text{if } N > N_Q \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Without loss of generality, the WC set can be understood as a quota. All elements are part of the wide competition. This configuration ensures that there will be no shortage of matches if the maximum matching is possible. In Figures 24a and 24b, this vertex is represented in gray.

5.3.3 Proxies

A proxy is an entity represented by a vertex, whose outgoing edge has unit capacity. There is always a proxy at the entrance of each element of the set where quotas are applied. This proxy ensures that each element is selected only once when it participates in multiple quota groups (e.g., a worker that belongs to both, disable and female subsets, or a worker that belongs to both, WC and quota group). In Figures 24a and 24b, this vertex is identified in dark green / dash-dotted.

5.3.4 Mapping

For a more formal description of the mapping, consider two sets U and V. The goal is to perform a matching M between U and V, minimizing the sum of the costs of the edges in M. Additionally, there is a set of quotas Q of arbitrary size $|Q|$, each quota is identified by q_i, where $0 < i \leq |Q|$. Each quota q_i represents the elements of a specific subset of U (the elements that belong to the q_i quota group, namely U_{q_i}). Each quota has an assigned number n_{q_i} that represents how many elements of q_i should be in the final matching. Each element of U, denoted as u_i, has its own proxy, called p_i. The set of all proxies is denoted by P, and a subset $P_{U_{q_i}}$ represents all proxies of the elements of U that belong to quota q_i.

The mapping always results in a multilayer graph containing six layers:

- **Relations between Layer 1 (*Source*) and Layer 2**: The flow always originates from the *Source*, located in Layer 1. It has $|Q|+1$ outgoing edges, connecting to the vertices of Layer 2, which represent each quota in Q and also the *wide competition* vertex, called *wc*. The weights of these edges are always zero, while the capacity is defined as n_{q_i} when connected to a q_i or n_{wc} (number of remaining matchings) when connected to *wc*.

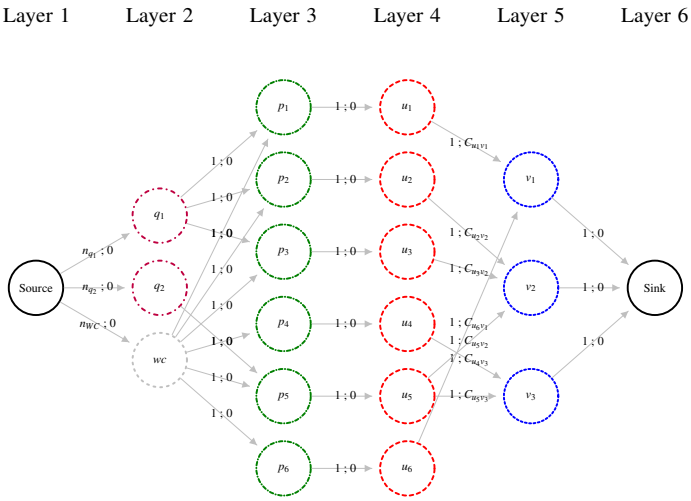
- **Relations between Layer 2 and Layer 3**: In Layer 3 are the proxies. Each vertex in Layer 2, q_i or *wc*, has an edge with capacity 1 and cost 0 connecting it to the corresponding proxy of the element of U contained in its subset. That is, every edge q_i will have an edge to the elements of $P_{U_{q_i}}$. Additionally, the vertex *wc* is connected to all vertices in Layer 3, always with capacity 1 and cost 0.

- **Relations between Layer 3 and Layer 4**: This pair of layers aims to prevent an element of U from being matched more than once. Every vertex P_{u_i} connects to its corresponding u_i, with zero cost and unit capacity.

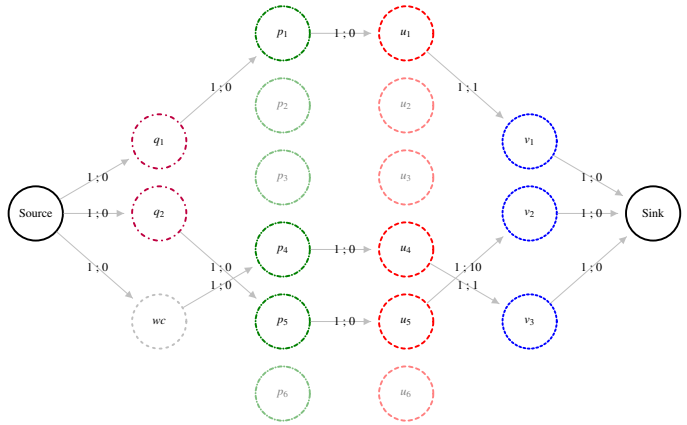
- **Relations between Layer 4 and Layer 5**: At this stage, matching costs are taken into account. The cost of the edges connecting the two sets is defined by the relationships between elements of U (Layer 4) and V (Layer 5), and the cost between element u_i and v_j is called $C_{u_i v_j}$. Additionally, the capacity remains 1.

- **Relations between Layer 5 and Layer 6 (*Sync*)**: These are the final edges and serve to ensure that the elements of V are matched only once, as well as to close the circuit cycle. Each element of V has an edge to the *Sink*, with capacity 1 and cost 0.

In Figure 24a, a visual representation of this mapping can be seen. Additionally, Figure 24b shows the solution to the same problem as Figure 24a if U_1, U_2 and U_3 belonged to Quota_1 and U_5 belonged to Quota_2.



(a) Example of a possible mapping of Fair Bipartite Matching.



(b) Solution of the problem from Figure 24a if u_1, u_2, and u_3 belonged to quota q_1 and u_5 belonged to quota q_2. The selected elements are those whose edge flows are not 0. The unselected elements are faded out. Total cost of 12.

Figure 24 – Examples of Fair Bipartite Matching mapping and its solution.

5.4 PROOFS

The objective of this proof is to demonstrate that the optimization respects the minimum quotas and minimizes costs, as well as to show that elements of a quota can participate in wide competition if it optimizes the matching.

5.4.1 Respecting Minimum Quotas

Lemma 1. *The total amount of flow passing through the vertices representing a quota q_i will not exceed n_{q_i}.*

To ensure that the predefined minimum quotas for the sets are fully respected during the matching process, it is crucial that the capacity configuration is well-defined. Each vertex in Layer 2 (quotas q_i and wide competition *wc*) is connected to the *Source* vertex with capacities defined as n_{q_i} for q_i and n_{wc} for *wc*. This guarantees that the maximum number of flows for each q_i is n_{q_i}, respecting the minimum quotas.

Furthermore, the MCMF problem finds the maximum flow that passes through the graph while respecting the edge capacities. The total amount of flow passing through the vertices representing the quotas q_i will not exceed n_{q_i}, ensuring that the minimum quotas are respected.

5.4.2 Cost Minimization

Lemma 2. *MCMF maximizes the flow in a graph, and among all maximum flows, it finds the one whose sum of edge costs is the smallest [12].*

Lemma 3. *The edges between the elements that optimize the MCMF represent the matchings between the elements of groups U and V [12].*

To minimize the costs associated with the matchings, the edges between the elements of U and V in Layers 4 and 5 have associated costs that represent the cost of matching. The MCMF algorithm minimizes the total cost of the flow by choosing the lowest-cost matchings between U and V.

Each edge between U and V has a capacity of 1, ensuring that each element is matched at most once. The minimization of the total cost will be based on selecting matchings that result in the lowest aggregate cost, respecting the capacity structure.

5.4.3 Respecting Matching Uniqueness

Lemma 4. *Every element in U and V has only one unit of outgoing flow, ensuring that no member of U and V is matched more than once.*

Every element in U has only one unit of outgoing flow, ensuring that no member of U is matched more than once. The same analysis applies to the vertices in V.

5.4.4 Theorem Formulation

Theorem 1. *If the filling of the quotas is possible, the mapping of a fair bipartite matching problem to an MCMF problem will fulfill the quotas, respect the use of each resource, and minimize costs.*

As discussed, the capacities and costs of the edges ensure that the quotas q_i are respected. The MCMF algorithm minimizes the total cost of the matching by selecting the lowest-cost edges between U and V. Additionally, no element of U and V will be matched more than once.

5.5 EXAMPLES

The objective of this section is to demonstrate various contexts in which the mapping of fairness in matching to the Minimum Cost Maximum Flow (MCMF) problem can be effectively applied.

For clarity and readability in the illustrations, the costs of the unsolved mappings will be suppressed.

5.5.1 Fairness in workers to positions allocation

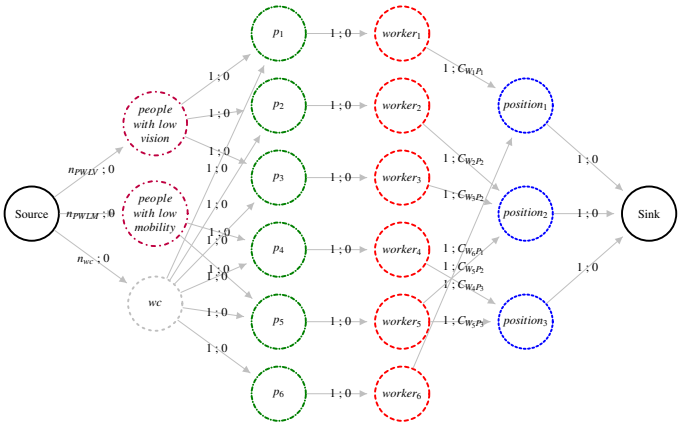
In the first scenario, we address the challenge of forming a diverse team where some workers have specific limitations, such as low vision or mobility. The objective is to assemble the most diverse and optimal team possible, guaranteeing that workers with low vision and low mobility are selected for the team. Each worker expresses their preferences for available job roles, and an objective evaluation method—such as an exam—assigns costs based on prior performance, with lower costs corresponding to higher performance to maintain a minimum-cost solution. Given that the jobs differ in nature, the evaluation process assigns distinct costs to each job-choice vertex, reflecting each worker's suitability for those roles.

The result that can be seen in Figure 25b is a minimum-cost solution that maximizes the diversity of attributes among the selected workers, ensuring both fairness and efficiency in the team formation process.

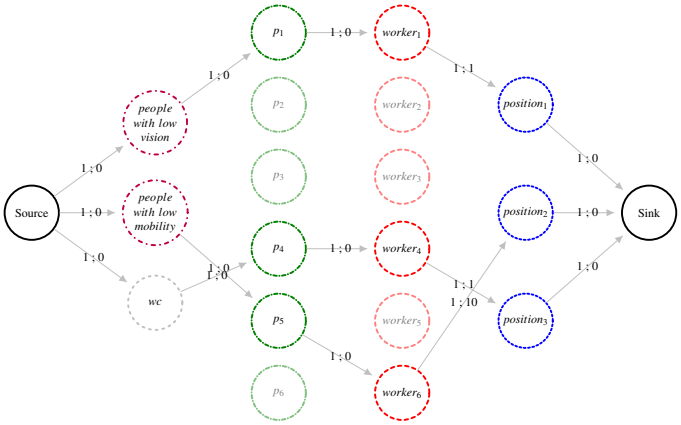
5.5.2 Fairness in server to services allocation

In the second example, we address a resource allocation problem within a network of services belonging to a single application. The goal is to assign these services to servers, each of which possesses distinct attributes, such as geolocation and backup methods. The objective is to ensure the most diverse and optimal distribution of services across the available servers. An evaluation process assigns costs based on factors like latency, representing the suitability of each service-server pair. This cost structure ensures that resources are allocated in a way that balances fairness and efficiency while maximizing server characteristics that minimize overall costs.

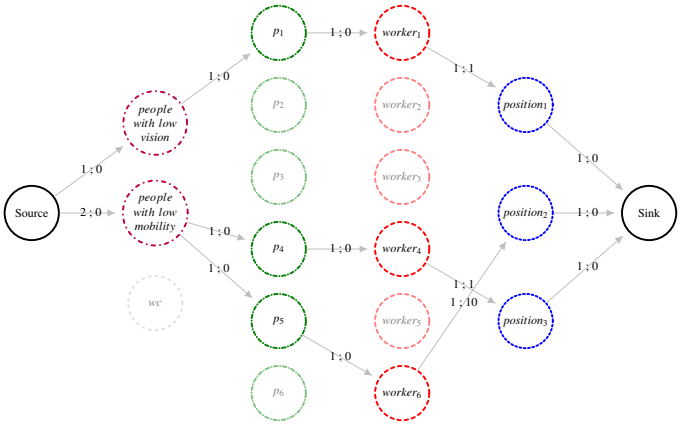
Ultimately, this approach leads to optimization of resource allocation by prioritizing diverse server attributes and minimizing latency costs, achieving an efficient and fair solution.



(a) Example of worker to positions mapping.

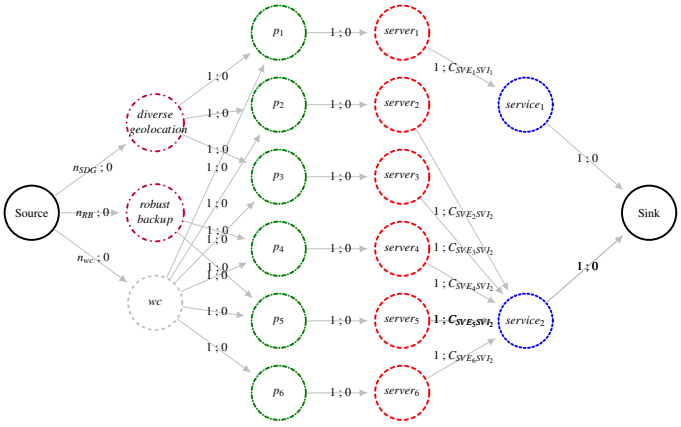


(b) Example of a solution for mapping workers to jobs in Figure 25a. In this example each quota has at least one assemble. Unselected workers and their proxies are faded out.

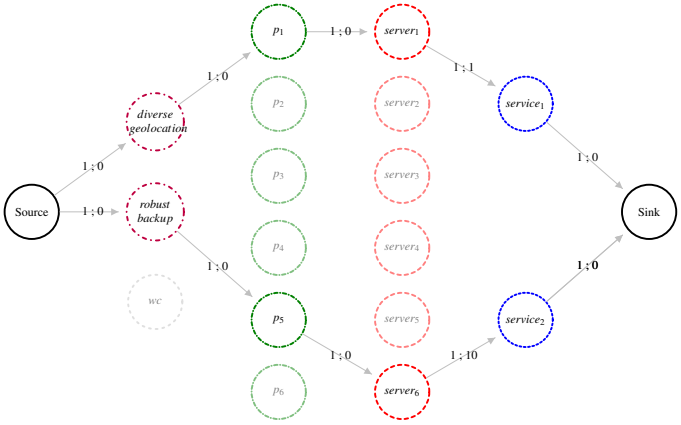


(c) Example of a solution for mapping workers to jobs in Figure 25a. In this example the ‘people with low mobility’ quota has a minimum of 2 assembles, thus no wide competition is assembled. Unselected workers and their proxies are faded out.

Figure 25 – Examples of worker to positions mapping and their solutions. (a) shows the initial mapping, (b) presents a solution with at least one assemble per quota, and (c) shows a solution where no wide competition is assembled.



(a) Server to services mapping.



(b) Solution to server to services mapping.

Figure 26 – Example of server to services allocation. (a) shows the initial mapping, and (b) presents the solution.

6 FRAMEWORK

Building upon the ideas presented in Chapter 4, we outline how an prototype Python framework was implemented. In this chapter, we demonstrate the construction process, focusing on a problem-agnostic approach. To maintain generality, some classes have been simplified, allowing us to emphasize the framework's core design principles rather than problem-specific details. The implementation of the framework is available on GitHub[1] [92].

6.1 REPRESENTING RELATIONS

To represent the rules of a grouping, as defined in Section 2.4.1, we provide the class *GroupRule* (see Script 6.1). The *GroupRule* provides methods to define the classes that can be part of a *Group*, the specifications, and also represents the rules of a specific *Group*, as defined in Script 6.1. In Section 6.4, we will provide examples of how to create these rules using the *GroupRule* class.

```
class GroupRule:
    # Sets the objective function (string or callable).
    def set_objective_function(self, function):
        # omitted code

    # Add classes to the group's relation and defines its min/max cardinality relation.
    def set_cardinality(self, cls, min_count, max_count):
        # omitted code

    # Adds a validator function for group validation.
    def add_validator(self, validator_fn):
        # omitted code

    # Adds a statistic function for the grouping.
    def add_statistic(self, statistic):
        # omitted code

    # Validates group, raises exception if invalid.
    def validate_or_raise(self, group):
        # omitted code

    # Validates group(s), returns True if valid.
    def validate(self, groups)
        # omitted code

    # Enables or disables stable matching.
    def set_stable_match(self, stable_match):
        # omitted code
```

Script 6.1 – *GroupRule* class that defines valid *Groups* and its specifications. Such as min/max cardinality of each relation, group validation functions, statistic functions, and object function.

[1] <https://github.com/RafaelGranza/Matching-Optimization-Framework>

Note that the *GroupRule* class supports defining cardinality constraints, validation functions, and statistics functions that can be used to evaluate the quality of the grouping. The validator functions are used to ensure that the grouping adheres to the specified rules, while the statistics functions can be used to gather information about the grouping, such as the number of members in each class or the average cost of the grouping.

The *Group* class is a flexible container for objects, as demonstrated in Script 6.2. It represents the elements (instances) that are grouped together, i.e., members of a group. Importantly, these objects can be instances of any class, and they are organized by their class type.

```
class Group:
    """
    A container of members, which can be any type of object.
    Members are organized by their class type.
    """

    # Adds a member to the group, allowing multiple instances.
    def add_member(self, *instances):
        # omitted code

    # Removes a member from the group, allowing multiple instances.
    def remove_member(self, *instances):
        # omitted code

    # Retrieves a dictionary of members divided by class type, optionally filtered by
        only one class type.
    def get_members(self, cls=None):
        # omitted code

    # Retrieves members of the group as a list, optionally filtered by class type.
    def get_members_as_list(self, cls=None):
       # omitted code

    # Returns a string representation of the group, showing all members and their types.
    def __repr__(self):
       # omitted code
```

Script 6.2 – *Group* Class. It represents the elements (instances) that are grouped together.

The complete implementations of Script 6.1 and Script 6.2 can be found at Appendix A.

6.2 SOLVERS

As described in Section 4.2.2, the framework is designed to accommodate various solvers, each tailored to specific grouping problems. These solvers are implemented as classes that inherit from the abstract class *Solver* (see Script 6.4). Different *Solvers* work with very different problems, and each problem may be solved by more than one available solver. Therefore, we provide an assigner to select the suitable solver for each problem.

6.2.1 Assigning Solvers

The Script 6.3 implements a simple solver assignment strategy equivalent to the flowchart of Figure 21. Returns the first solver capable of handling the problem from a priority list. The high-priority algorithms are the deterministic ones.

```
class Assigner:

    solvers: List[Solver] = [
        HungarianAlgorithm,
        BinSearchHungarianAlgorithm,
        StableMarriage,
        FairBipartiteMatching,
        GeneticAlgorithm, # Metaheuristic solver
        # Add more solvers here as needed (e.g., SimulatedAnnealing)
    ]

    def choose_solver(self, group_rule: GroupRule) -> Solver:
        """
        Choose the best solver based on the group members and their cardinality.
        """
        for solver in self.solvers:
            if solver.can_solve(group_rule):
                return solver
        raise ValueError("No suitable solver found for the given group.")

    def add_solver(self, solver: Type[Solver]):
        """
        Add a new solver to the list of available solvers.
        """
        self.solvers.append(solver)

    def remove_solver(self, solver: Type[Solver]):
        """
        Remove a solver from the list of available solvers.
        """
        self.solvers.remove(solver)
```

Script 6.3 – Assigner Class.

The proposed design ensures a seamless integration of new solvers, as the responsibility of verifying whether a solver can handle a given problem lies within the solver itself. The order in which solvers are added, and consequently the sequence in which they are evaluated, dictates the preference when multiple solvers are capable of addressing the same problem.

The assigner iterates through a predefined list of solvers, selecting the first one that successfully resolves the given problem. The Genetic Algorithm is positioned as the last resort, ensuring that it is only employed when no specialized solver is available.

6.2.2 Adding Solvers

A new solver can be implemented by creating a class that inherits from *Solver* (see Script 6.4) and by overriding the necessary virtual functions. Once implemented, the solver must be

added to the assigner's list, as demonstrated in Script 6.3. In Section 6.2.3, four default solvers are introduced.

```
class Solver:
    """
    Abstract class for solvers. All solvers should inherit from this class.
    """

    @staticmethod
    def can_solve(group_rule: GroupRule):
        """
        Check if the solver can solve the given group_rule.
        """
        raise NotImplementedError("Subclasses should implement this method.")

    @staticmethod
    def solve(group_rule: GroupRule, instances=List[object]):
        """
        Solve the given group.
        """
        raise NotImplementedError("Subclasses should implement this method.")

    @staticmethod
    def solve(group_rule: GroupRule, groups: List[Group]):
        """
        Solve the given group.
        """
        raise NotImplementedError("Subclasses should implement this method.")
```

Script 6.4 – Solver Class.

6.2.3 Implemented Solvers

In this section, we describe the different solvers that were implemented to address the grouping problems. Each solver is chosen based on its suitability for the problem at hand, ranging from optimization-based techniques like the Hungarian Algorithm to more heuristic-based approaches such as Genetic Algorithms. We also delve into a solver designed for specific cases like the Stable Marriage problem, which ensures a stable matching between two groups.

It is important to notice that for this work, it is enough to demonstrate possibilities, indicating an arbitrary amount of solvers can be implemented and added to this framework in the future. The solvers implemented in this framework are available at Appendices B, C, D, E, and F.

6.2.3.1 Metaheuristic

To implement the genetic algorithm, we utilized the DEAP library [79], which provides a flexible framework for evolutionary algorithms in Python. In this implementation, individuals are represented as lists of booleans, and for matrix-based problems, we use a compressed matrix representation as a flat list.

DEAP offers customizable genetic operators and supports user-defined fitness functions, allowing constraints to be incorporated directly. The default mutation and crossover operators

from DEAP are used, enabling efficient exploration of large search spaces. This solver is particularly suitable for complex matching problems where traditional optimization algorithms are impractical and approximate solutions are sufficient.

6.2.3.2 Hungarian Algorithm

The Hungarian Algorithm, also known as the Kuhn-Munkres algorithm, was implemented to solve assignment problems where the goal is to find an optimal, minimum-cost matching between two sets of objects. The algorithm operates in polynomial time, making it an efficient choice for solving balanced bipartite matching problems [42].

In this implementation, we followed the standart approach of constructing a cost matrix, where each element represents the "cost" or "penalty" of matching a pair of objects. The algorithm then attempts to minimize the total cost by identifying the optimal set of pairings that leads to the lowest overall value. This is particularly suited for scenarios where the goal is not only to match but to solve a single objective, such as minimizing total distance or maximizing total compensation in the distribution of resources. The Hungarian Algorithm is deterministic, guaranteeing an optimal solution for these types of problems.

6.2.3.3 Binary Search and Hungarian Algorithm

The combination of binary search and the Hungarian Algorithm provides an efficient approach to solving matching problems where constraints on edge rankings play a crucial role. This method is particularly useful when the objective is to find a maximum matching while ensuring that the lowest-ranked edge in the matching is as high as possible.

The approach consists of performing a binary search on the minimum allowable edge ranking. For a given threshold value T, all edges with rankings lower than T are temporarily removed from consideration, leaving only edges with rankings $\geq T$. The Hungarian Algorithm is then applied to find the optimal matching under these constraints. The binary search then adjusts the threshold until the maximum possible minimum ranking is achieved while maintaining a valid matching.

More formally, the algorithm follows these steps:

1. Define the search space by setting an initial range $[L,R]$, where L is the lowest ranking in the graph and R is the highest ranking.

2. Perform binary search on T:

    - Set $T = \frac{L+R}{2}$;
    - Remove all edges with rankings lower than T;
    - Apply the Hungarian Algorithm to determine the maximum matching under the new graph constraints;

- If a valid matching exists, adjust L to try increasing T; otherwise, adjust R.

3. Repeat until convergence, returning the best feasible matching where the lowest-ranked edge is maximized.

This technique ensures that the worst-case edge ranking in the matching is as high as possible, making it particularly useful in applications where fairness or quality guarantees are required. By leveraging binary search, the solution efficiently narrows down the feasible rankings, ensuring that the Hungarian Algorithm is executed only on relevant subsets of the graph, thereby improving computational efficiency in large-scale instances.

6.2.3.4 Stable Marriage Algorithm

The Stable Marriage Algorithm, based on the Gale-Shapley Deferred Acceptance algorithm, was implemented to solve matching problems where stability between two groups is the primary concern. Stability in this context refers to a matching where no pair of individuals would prefer each other over their current matches, thus preventing any potential disruptions to the arrangement.

For the implementation, we focused on creating an efficient solution for the stable matching problem, where each group provides a preference list, and the algorithm iteratively matches individuals based on their rankings. We applied the "men-proposing" variant of the Stable Marriage Algorithm, ensuring that once a match is formed, it is either retained or improved upon without destabilizing previous matches.

This algorithm is particularly effective in applications where stability are more important than optimization, such as job candidate assignment or college admissions. The result is a matching that respects individual preferences while maintaining stability across the entire system.

6.2.3.5 Fair Bipartite Matching

The theoretical foundations for this solver are presented in Chapter 5. For the graph-related implementation, we used both the networkx and igraph libraries [93, 94], which offer tools for graph manipulation and for solving minimum-cost maximum-flow problems.

6.3 SOLVE

For the purpose of this framework, we provide a simple *solve* function that selects the appropriate solver based on the problem's characteristics and solves the problem. The *solve* function provides a direct interface for solving grouping problems. It accepts either a *GroupRule* and a list of valid *Groups*, or a *GroupRule* and a list of instances. The function internally uses the *Assigner* to select the appropriate solver and applies it to the problem, abstracting away the manual selection and invocation of solvers. See Script 6.5 for details.

```
def solve(gr: GroupRule, instances: List[object]):
    assigner = Assigner()
    solver = assigner.choose_solver(gr)

    if all(isinstance(inst, Group) for inst in instances):
        return solver.solve_from_valid_groups(gr, instances)

    return solver.solve_from_instances(gr, instances)
```

Script 6.5 – Solve.

6.4 PROBLEM EXAMPLES

To illustrate the flexibility of the framework in solving various grouping problems, this section presents four distinct problems, each one with its unique requirements. Note that whilst the classes and instances are manually created in the examples, in a real application this could come from a database or other data source, and the framework would still be able to handle it seamlessly.

6.4.1 Job Assignment

In this example, the goal is to match workers to jobs while minimizing a cost metric (e.g., skill) as described at Section 3.1.1. In Script 6.6, we configure the framework with a cost function that evaluates the expense of assigning specific workers to jobs, with a one-to-one assignment.

```
from src.group import Group, GroupRule
import random
from typing import List, Type
import itertools
from src.solve import solve

# Define the Classes for the Example
class Worker:
    def __init__(self, name):
        self.name = name
        self.skills = random.randint(1, int(1e6)) # Simulating worker skill level
    def __repr__(self): return f"Worker({self.name})"
class Job:
    def __init__(self, title):
        self.title = title
        self.skills = random.randint(1, int(1e6)) # Simulating skills required
    def __repr__(self): return f"Job({self.title})"

# Define the Statistic Function
def skill_allignment(members: dict[Type, List]):
    workers = members.get(Worker, [None])[0]
    jobs = members.get(Job, [None])[0]
    return abs(workers.skills - jobs.skills)

# Create instances
workers = [
    Worker("Alice"), Worker("Bob"), Worker("Charlie"), Worker("Diana"),
```

```
    Worker("Eve"), Worker("Frank"), Worker("Grace"), Worker("Hank"),
    Worker("Ivy"), Worker("Jack"), Worker("Karen"), Worker("Leo"),
]
jobs = [
    Job("Painting"), Job("Driving"), Job("Plumbing"), Job("Cleaning"), Job("Gardening")
]

# Create Groups
groups = [Group().add_member(*comb) for comb in itertools.product(workers, jobs)]

gr = GroupRule()                    # Define the Group Rule
gr.set_cardinality(Worker, 1, 1)    # exactly 1 worker per job
gr.set_cardinality(Job, 1, 1)       # exactly 1 job per worker
gr.add_statistic(skill_allignment) # adding the skill alignment statistic
gr.set_objective_function("minimize_sum_of_single_statistic")

print("Answer: ", solve(gr, groups)) # Optimizing from a list of Groups
```

Script 6.6 – Creating and solving the Job Assignment Problem using this framework.

6.4.2 Stable Marriage

This example in Script 6.7 demonstrates stable pairing, where two groups (e.g., males and females) are matched based on mutual preferences, further explanation can be found at Section 3.1.4. The framework allows defining preference-based rules to ensure a stable solution where no pairs would prefer a different match. The fitness function of the following function is shown in Algorithm 3 in Section 4.1.3, and its implementation is builtin, there is only need to configure the preferences and set stable matching.

```
from src.group import Group, GroupRule
import random
from typing import List, Type
from src.solve import solve

# Define the Classes for the Example
class Man:
    def __init__(self, name, preferences=None):
        self.name = name
        self.preferences = preferences if preferences is not None else []
    def add_preference(self, preference):
        if preference not in self.preferences:
            self.preferences.append(preference)
    def __repr__(self): return f"Man({self.name})"

class Woman:
    def __init__(self, name, preferences=None):
        self.name = name
        self.preferences = preferences if preferences is not None else []
    def add_preference(self, preference):
        if preference not in self.preferences:
            self.preferences.append(preference)
    def __repr__(self): return f"Woman({self.name})"


# Define a function to validate if the preferences are repected:
```

```
# The Stabel Matching does not allow statistics, so this is a validator function.
def stable_matching_validator(members):
    males = members.get(Man, [])
    females = members.get(Woman, [])
    male_partner = {m: f for m, f in zip(males, females)}
    female_partner = {f: m for m, f in zip(males, females)}

    for m in males:
        for w_name in preferences[m.name]:
            w = next((f for f in females if f.name == w_name), None)
            if w is None:
                continue
            if female_partner[w] != m:
                current_w = male_partner[m]
                current_m = female_partner[w]

                m_prefers_w = preferences[m.name].index(w.name) < preferences[m.name].
                    index(current_w.name)
                w_prefers_m = preferences[w.name].index(m.name) < preferences[w.name].
                    index(current_m.name)

                if m_prefers_w and w_prefers_m:
                    return False
    return True


# Create instances
men = {
    "John":   Man("John"),   "Paul":  Man("Paul"),  "Mike":    Man("Mike"),
    "George": Man("George"), "Ringo": Man("Ringo"), "Pete":    Man("Pete"),
    "Brian":  Man("Brian"),  "Roger": Man("Roger"), "Freddie": Man("Freddie")
}

women = {
    "Mary":  Woman("Mary"),  "Patricia": Woman("Patricia"), "Susan":   Woman("Susan"),
    "Linda": Woman("Linda"), "Karen":    Woman("Karen"),    "Jessica": Woman("Jessica"),
    "Sarah": Woman("Sarah"), "Jennifer": Woman("Jennifer"), "Nancy":   Woman("Nancy")
}

# Define preferences
men_preferences = {
    "John":    [women["Mary"],     women["Linda"],    women["Susan"]   ],
    "Paul":    [women["Linda"],    women["Mary"],     women["Susan"]   ],
    "Mike":    [women["Susan"],    women["Mary"],     women["Linda"]   ],
    "George":  [women["Patricia"], women["Jennifer"], women["Jessica"] ],
    "Ringo":   [women["Jennifer"], women["Patricia"], women["Jessica"] ],
    "Pete":    [women["Jessica"],  women["Patricia"], women["Jennifer"]],
    "Brian":   [women["Sarah"],    women["Karen"],    women["Nancy"]   ],
    "Roger":   [women["Karen"],    women["Sarah"],    women["Nancy"]   ],
    "Freddie": [women["Nancy"],    women["Sarah"],    women["Karen"]   ],
}

woman_preferences = {
    "Mary":     [men["John"],    men["Paul"]   ],
    "Linda":    [men["Paul"],    men["Mike"]   ],
    "Susan":    [men["Mike"],    men["George"] ],
    "Patricia": [men["George"],  men["Ringo"]  ],
    "Jennifer": [men["Ringo"],   men["Pete"]   ],
    "Jessica":  [men["Pete"],    men["Brian"]  ],
```

```
    "Sarah":    [men["Brian"],   men["Roger"]  ],
    "Karen":    [men["Roger"],   men["Freddie"]],
    "Nancy":    [men["Freddie"], men["John"]   ],
}

for name, man in woman_preferences.items():
    women[name].add_preference(man)

for name, woman in men_preferences.items():
    men[name].add_preference(woman)

instances = list(men.values()) + list(women.values())


# Create the Group Rule
gr = GroupRule()
gr.set_cardinality(Woman, 1, 1)
gr.set_cardinality(Man, 1, 1)
gr.set_stable_match(True)                    # Enables stable matching
gr.add_validator(stable_matching_validator) # Adds the validator function

# Optimizing from a list of Instances, the groups are built automatically.
print("Answer: ", solve(gr, instances))
```

Script 6.7 – Creating and solving the Stable Marriage Problem using this framework.

6.4.3 Timetable Scheduling

For complex scheduling scenarios, such as assigning professors, rooms, lectures, and times, we use a genetic algorithm. This example defines a fitness function to solve resource allocation while meeting scheduling constraints.

```
from src.group import Group, GroupRule
import itertools
from src.solve import solve


# Define the Classes for the Example
class Professor:
    def __init__(self, name): self.name = name
    def __repr__(self): return f"Professor({self.name})"

class Room:
    def __init__(self, name): self.name = name
    def __repr__(self): return f"Room({self.name})"

class Cohort:
    def __init__(self, name): self.name = name
    def __repr__(self): return f"Cohort({self.name})"

class TimeWindow:
    def __init__(self, slot): self.slot = slot
    def __repr__(self): return f"TimeWindow({self.slot})"

class Subject:
    def __init__(self, name): self.name = name
    def __repr__(self): return f"Subject({self.name})"
```

```
# Define an objective function to evaluate the groups
def objective_function(groups):
    professors_by_time = dict()
    rooms_by_time = dict()
    score = 0
    for group in groups:
        elements = group.get_members()
        time = elements[TimeWindow][0]
        professor = elements[Professor][0]
        room = elements[Room][0]

        if time not in professors_by_time:
            professors_by_time[time] = set()
        if time not in rooms_by_time:
            rooms_by_time[time] = set()

        # Check for conflicts in professors and rooms at the same time
        if professor in professors_by_time[time]:
            score += -1e6 # Conflict penalty for professor
        else:
            score += 1e6 # Reward for only one professor at a given time
        if room in rooms_by_time[time]:
            score += -1e6 # Conflict penalty for room
        else:
            score += 1e6 # Reward for only one room at a given time

        professors_by_time[time].add(professor)
        rooms_by_time[time].add(room)

    return score


# Create instances and groups for the problem
professors =   [Professor("ProfA"), Professor("ProfB"), Professor("ProfC")]
rooms =        [Room("Room1"), Room("Room2"), Room("Room3")]
cohorts =      [Cohort("Cohort1"), Cohort("Cohort2"), Cohort("Cohort3")]
time_windows = [TimeWindow("8h"), TimeWindow("10h"), TimeWindow("12h")]
subjects =     [Subject("Math"), Subject("History"), Subject("Science")]

instances =  [professors, rooms, cohorts, time_windows, subjects]
all_groups = [Group().add_member(*comb) for comb in itertools.product(*instances)]

# Define the Group Rule
gr = GroupRule()
gr.set_cardinality(Professor, 1, 1)
gr.set_cardinality(Room, 1, 1)
gr.set_cardinality(Cohort, 1, 1)
gr.set_cardinality(TimeWindow, 1, 1)
gr.set_cardinality(Subject, 1, 1)
gr.set_objective_function(objective_function)


# solve from a list of Groups
print(solve(gr, all_groups))
```

Script 6.8 – Creating and solving the Timetable Scheduling Problem using this framework.

To see more information about resource allocation problems, see Section 3.1.2.

6.4.4 Fair Bipartite Matching

An algorithm for the Fair Bipartite Matching Problem was implemented, as detailed in Chapter 5. This implementation allows the framework to address scenarios in which minimum quotas must be taken into account during grouping or resource allocation.

Script 6.9 illustrates an example similar to Script 6.6, with the main difference being the inclusion of minimum quotas. Notably, the example from Script 6.9 would be assigned to the *HungarianAlgorithm* solver, in case the quotas were not defined. This example corresponds to the scenario represented in Section 5.5.1.

```
from src.group import GroupRule
import random
from typing import List, Type
from src.solve import solve

class Worker:
    def __init__(self, ID, Disabilities):
        self.ID = ID
        self.skills = random.randint(1, int(1e6)) # Simulating skills
        self.Disabilities = Disabilities
    def __repr__(self):
        return f"Worker(ID={self.ID}, Disabilities='{self.Disabilities}')"
class Job:
    def __init__(self, ID):
        self.ID = ID
        self.skills = random.randint(1, int(1e6)) # Simulating skills required
    def __repr__(self):
        return f"Job({self.ID})"

# Define the Statistic Function
def skill_allignment(members: dict[Type, List]):
    workers = members.get(Worker, [None])[0]
    jobs = members.get(Job, [None])[0]
    return abs(workers.skills - jobs.skills)

# Create instances for the problem
def gen_jobs(qnt):
    # Generate a list of Job instances
    list = []
    for i in range(qnt):
        list.append(Job(ID=i))
    return list

def gen_workers(qnt):
    # Generate a list of Worker instances, with some having disabilities
    list = []
    disabilities = ["Low Visibility", "Low Mobility", "None"]
    for i in range(0, int(qnt*0.8)):
        list.append(Worker(
            ID=i,
            Disabilities=disabilities[2]
        ))
    for i in range(int(qnt*0.8), qnt):
        list.append(Worker(
            ID=i,
            Disabilities=disabilities[random.randint(0,len(disabilities)-1)]
```

```
        ))
    return list

instances = gen_jobs(3) + gen_workers(6)

# Define the problem rules
gr = GroupRule()
# Two slots for both low mobility and low visibility
gr.quotas = {"Disabilities":[["Low Visibility", 2],["Low Mobility", 2]]}
gr.set_cardinality(Job, 1, 1)           # Each group must have exactly 1 job
gr.set_cardinality(Worker, 1, 1)        # Each group must have exactly 1 worker
gr.add_statistic(skill_allignment)      # Add the skill alignment statistic
# Use the sum of the statistic as the objective function
gr.set_objective_function("minimize_sum_of_single_statistic")

# solve from a list of instances
print("Answer: ", solve(gr, instances))
```

Script 6.9 – Creating and solving a Fair Matching Problem using this framework.

7 CONCLUSION

This work presented a framework designed to address a wide range of grouping problems, with a focus on flexibility, accessibility, and practical applications for industry. Through a modular design and support for various optimization techniques, including both exact and heuristic methods, the framework provides an effective tool for developers tackling complex grouping problems across diverse contexts.

7.1 CONTRIBUTIONS

This work makes several contributions to the field of grouping problems, each designed to enhance the flexibility, applicability, and efficiency of solving these problems in real-world scenarios. The contributions outlined below highlight the definition of a metaheuritics to solve grouping problem, the development of a framework that integrates various optimization techniques, a new algorithm for the minimum quota bipartite matching problem and gathering information about the current state of matching problems articles.

7.1.1 Metaheuristic

A contribution of this work is the mapping of grouping problems to a genetic algorithm (GA) approach. By developing a structured method for representing grouping problems within the genetic algorithm framework, this work provides a foundation for solving complex, NP-hard problems that may not be feasible with exact algorithms. This contribution enables approximate solutions through GA for various types of grouping problems, expanding the framework's capacity to handle cases where computational efficiency and flexibility are prioritized over exact solutions.

7.1.2 Framework

The proposed framework represents a flexible and extensible tool designed for a variety of matching problems. By allowing users to define problem-specific constraints, costs, and preferences, the framework accommodates both standard and customized matching configurations. This adaptability enables its application across multiple domains, from resource allocation to scheduling, and makes it suitable for developers with varying levels of expertise in optimization.

7.1.2.1 Limitations

Despite its versatility, the framework has some limitations. One significant limitation is the requirement for users to implement their own fitness function to define problem-specific optimization criteria. This requires familiarity with both the framework and the underlying optimization principles, which may be a barrier for some users.

Additionally, the framework currently provides only three solvers. Although these cover a range of matching problems, users must add new solvers if they require support for problems with unique "statistics" or specific geometries not handled by the existing options. Expanding the solver library could enhance the framework's applicability to a broader array of problems.

7.1.3 Fair Matching Algorithm

The present work introduced an approach to the bipartite matching problem with the inclusion of minimum quotas, aiming to promote fairness in resource allocation. The proposed mapping for the Minimum Cost Maximum Flow (MCMF) problem, adapted to consider quotas, proved valid in the experiments conducted.

In this work, we focus on the fair allocation of resources in scenarios where supply (U) exceeds demand (V). Without loss of generality, swapping the sets U and V allows deploying our strategy in cases where demand exceeds supply.

Nevertheless, it is important to highlight that there are several areas to be explored in future work. Verifying the effectiveness and significance of the quotas in different contexts is an open field, requiring more in-depth analyses and broader experiments. Additionally, the implementation must be subjected to rigorous testing to ensure the correctness of the code, as well as performance optimization.

Other research directions include investigating alternative approaches for the definition and application of quotas, as well as exploring machine learning techniques to optimize quota selection in different scenarios. Refining the proposed model and analyzing its applicability across various domains also represent valuable opportunities for future research.

In summary, this work is an initial step toward understanding and applying quotas in the context of bipartite matching, but there is a vast area to be explored in pursuit of a more comprehensive understanding and improvement of the proposed approach.

7.1.3.1 Advantages

One of the fundamental advantages of this approach is the ability to employ any MCMF algorithm, even those that are not exact. This flexibility is particularly beneficial, as the different existing implementation methods place varying importance on the trade-off between execution time and solution accuracy.

Furthermore, there is the opportunity to use distributed techniques to solve the MCMF problem, broadening the available options for addressing it. Depending on the chosen algorithm, it is not necessary to have all relationships between the elements of each set precalculated, which can reduce memory usage and processing time.

In summary, the approach offers great flexibility, allowing it to be computed by various algorithms, thus providing a wide range of options for solving the problem.

7.1.4 Review of Recent Developments in Matching Problems and Algorithms

This work also conducted a preliminary review of recent advancements in matching problems, focusing on contributions made since 2021. Through a comprehensive search of relevant studies, we identified developments in matching algorithms and problem types. After filtering the studies by language and relevance, we examined the most prominent topics that emerged from the remaining works. This review highlights the continued evolution of the field and contributes to the understanding of cutting-edge solutions and their potential applications in industry.

7.2 FUTURE WORK

Future work could focus on several areas to enhance the framework's capabilities and usability. One key direction is expanding the library of built-in solvers to support a broader range of matching problems, thereby reducing the need for users to implement custom solvers for specific configurations. Another important improvement involves automating the definition of fitness functions or providing built-in options that address common optimization criteria, which would make the framework more accessible to non-expert users. In addition, optimizing the metaheuristic solver to handle large-scale datasets more efficiently, as well as incorporating support for multi-criteria optimization in fair matching problems, could significantly increase the framework's robustness and versatility.

The Set Cover Framework (SCF) could also be extended to tackle clustering problems by mapping them into grouping problems. This would enable the framework to address scenarios where elements need to be grouped based on similarity or other clustering criteria. Furthermore, we plan to explore fairness concepts beyond bipartite settings. For instance, in timetabling problems, it may be necessary to match rooms, modules, lectures, and student schedules under fairness constraints. Lastly, we will investigate fairness in bipartite graphs based on maximum quota principles, exploring how such constraints affect the structure and solutions of fair matching problems.

BIBLIOGRAPHY

1 REN, J. e. a. Matching algorithms: Fundamentals, applications and challenges. *IEEE Transactions on Emerging Topics in Computational Intelligence*, IEEE, v. 5, n. 3, p. 332–350, 2021. Citado 7 vezes nas páginas 9, 15, 31, 33, 34, 35, and 38.

2 KARP, R. M. Reducibility among combinatorial problems. In: MILLER, R. E.; THATCHER, J. W. (Ed.). *Complexity of Computer Computations*. [S.l.]: Plenum Press, 1972. p. 85–103. Citado 4 vezes nas páginas 15, 23, 38, and 47.

3 CORMEN, T. H. et al. *Introduction to Algorithms*. [S.l.]: MIT Press, 2022. Citado 4 vezes nas páginas 17, 18, 19, and 22.

4 BONDY, J.; MURTY, U. *Graph theory with applications*. [S.l.]: North Holland, 1976. Citado 4 vezes nas páginas 17, 18, 19, and 22.

5 WEST, D. B. *Introduction to Graph Theory*. [S.l.]: Prentice Hall, 2001. Citado 2 vezes nas páginas 19 and 32.

6 DIESTEL, R. *Graph Theory*. [S.l.]: Springer, 2017. Citado 2 vezes nas páginas 19 and 32.

7 HOPCROFT, J. E.; KARP, R. M. n. 5. *SIAM Journal on Computing*, v. 2, n. 4, p. 225–231, 1973. Citado na página 20.

8 SANKAR, G. S. et al. Matchings with group fairness constraints: Online and offline algorithms. *arXiv*, 2021. Citado na página 20.

9 KORTE, B.; VYGEN, J. *Combinatorial Optimization: Theory and Algorithms*. 6. ed. [S.l.]: Springer, 2018. Citado na página 21.

10 FORD, L. R.; FULKERSON, D. R. *Maximal flow through a network*. [S.l.]: Canadian journal of Mathematics, 1956. Citado 2 vezes nas páginas 21 and 32.

11 AHUJA, R. K.; MAGNANTI, T. L.; ORLIN, J. B. *Network flows: theory, algorithms, and applications*. [S.l.]: Prentice Hall, 1993. Citado 4 vezes nas páginas 21, 32, 47, and 48.

12 EDMONDS, J.; KARP, R. M. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, ACM, v. 19, n. 2, p. 248–264, 1972. Citado 3 vezes nas páginas 21, 48, and 53.

13 TARJAN, R. E. Dynamic trees as an efficient representation of network flow algorithms. *Mathematical Programming*, Springer, v. 78, p. 169–177, 1997. Citado 3 vezes nas páginas 21, 47, and 48.

14 AKIDAU, T.; AL. et. Millwheel: Fault-tolerant stream processing at internet scale. In: VLDB. *Proceedings of the VLDB Endowment*. [S.l.], 2013. v. 6(11), p. 1033–1044. Citado na página 22.

15 CHEN, L. et al. Maximum flow and minimum-cost flow in almost-linear time. In: *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. [S.l.: s.n.], 2022. p. 612–623. Citado na página 22.

16 ALON, N.; COHEN, Y.; AL. et. Decentralized network algorithms for min-cost flow: Convergence and application to blockchain. *Distributed Computing*, Springer, v. 32, n. 1, p. 53–71, 2019. Citado na página 22.

17 BRANDAO, F. G.; AL. et. Quantum speed-ups for solving semidefinite programs. *Proceedings of the Annual ACM Symposium on Theory of Computing*, v. 51, p. 141–150, 2019. Citado na página 22.

18 HOCHBAUM, D. S. Approximation algorithms for np-hard problems. *ACM Computing Surveys*, v. 28, n. 1, p. 165–164, 1997. Citado 2 vezes nas páginas 23 and 24.

19 LI, M.; TROMP, J.; VITÁNYI, P. M. Mapping onto the exact set cover problem in computational biology. *Theoretical Computer Science*, v. 287, n. 1, p. 1–17, 2002. Citado na página 23.

20 CHVÁTAL, V. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, v. 4, n. 3, p. 233–235, 1979. Citado 2 vezes nas páginas 23 and 24.

21 JIA, X.; LIN, D.; AL. et. Set covering algorithms for network design. *IEEE Transactions on Networking*, v. 10, n. 6, p. 968–977, 2002. Citado na página 23.

22 MEGUERDICHIAN, S. et al. Exposure in wireless ad-hoc sensor networks. In: *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*. [S.l.: s.n.], 2001. p. 139–150. Citado na página 24.

23 GAREY, M. R.; JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. [S.l.]: W. H. Freeman and Company, 1979. Citado 3 vezes nas páginas 24, 38, and 40.

24 VAZIRANI, V. V. *Approximation algorithms*. [S.l.]: Springer Science & Business Media, 2001. Citado na página 24.

25 MOTWANI, R.; RAGHAVAN, P. *Randomized Algorithms*. [S.l.]: Cambridge University Press, 1995. Citado na página 24.

26 AMBLER, S. W. *Object-oriented modeling with UML: A practical approach*. [S.l.]: Cambridge University Press, 2002. Citado na página 24.

27 FOWLER, M. *Patterns of Enterprise Application Architecture*. [S.l.]: Addison-Wesley, 2003. Citado 2 vezes nas páginas 24 and 32.

28 LARMAN, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. [S.l.]: Pearson Education, 2004. Citado 3 vezes nas páginas 24, 27, and 32.

29 BERNSTEIN, J. *Object-Relational Mapping (ORM): MongoDB, MongooseJS and NodeJS By Example*. [S.l.]: Jason Bernstein, 2009. Citado na página 25.

30 HOLLAND, J. H. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. [S.l.]: MIT press, 1992. Citado 4 vezes nas páginas 27, 28, 29, and 45.

31 NEWMAN, M. Networks. 2018. Citado na página 31.

32 GUSFIELD, D.; IRVING, R. W. *The stable marriage problem: structure and algorithms*. [S.l.]: MIT press, 1989. Citado 2 vezes nas páginas 31 and 32.

33 DEMANGE, G.; GALE, D.; SOTOMAYOR, M. Multi-item auctions. *Journal of Political Economy*, JSTOR, v. 94, n. 4, p. 863–872, 1986. Citado 2 vezes nas páginas 31 and 32.

34 WANG, J.; ZHANG, Y. Quantum matching pursuit: A quantum algorithm for sparse representations. *Quantum Information Processing*, v. 20, n. 3, p. 1–15, 2021. Citado 2 vezes nas páginas 31 and 36.

35 LI, H.; WANG, Y. Self-supervised learning of visual graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022. Citado 2 vezes nas páginas 31 and 36.

36 ZHOU, K.; LIU, J. Cross-modal graph matching network for image-text retrieval. *Proceedings of CVPR*, 2021. Citado 2 vezes nas páginas 31 and 36.

37 FERNANDEZ, M.; JONES, R. Solving the humanitarian multi-trip cumulative capacitated routing problem via a grouping metaheuristic algorithm. *European Journal of Operational Research*, 2023. Citado 2 vezes nas páginas 31 and 37.

38 MANLOVE, D. F. *Algorithmics of matching under preferences*. [S.l.]: World Scientific, 2013. Citado 3 vezes nas páginas 31, 32, and 33.

39 ROTH, A. E.; SOTOMAYOR, M. Two-sided matching: A study in game-theoretic modeling and analysis. *Econometric Society Monographs*, Cambridge University Press, v. 18, 1990. Citado na página 31.

40 SCHRIJVER, A. *Combinatorial Optimization: Polyhedra and Efficiency*. [S.l.]: Springer Science & Business Media, 2003. Citado na página 32.

41 GALE, D.; SHAPLEY, L. S. College admissions and the stability of marriage. *The American Mathematical Monthly*, JSTOR, v. 69, n. 1, p. 9–15, 1962. Citado 3 vezes nas páginas 32, 33, and 45.

42 KUHN, H. W. *The Hungarian method for the assignment problem*. [S.l.]: Naval Research Logistics Quarterly, 1955. Citado 4 vezes nas páginas 32, 33, 45, and 61.

43 ROTH, A.; SOTOMAYOR, M. Two-sided matching. In: *Handbook of game theory with economic applications*. [S.l.]: Elsevier, 1992. v. 1, p. 485–541. Citado na página 33.

44 BECKER, G. A theory of marriage: Part i. *Journal of Political Economy*, v. 81, n. 4, p. 813–846, 1973. Citado na página 33.

45 BERGSTROM, T.; BAGNOLI, M. Courtship as a waiting game. *Journal of Political Economy*, v. 101, n. 1, p. 185–202, 1993. Citado na página 33.

46 ROSEN, S. The economics of superstars. *The American Economic Review*, v. 71, n. 5, p. 845–858, 1981. Citado na página 33.

47 ROTH, A. Stability and polarization of interests in job matching. *Econometrica: Journal of the Econometric Society*, p. 47–57, 1984. Citado na página 33.

48 KREMER, M. The o-ring theory of economic development. *The Quarterly Journal of Economics*, v. 108, n. 3, p. 551–575, 1993. Citado na página 33.

49 GABAIX, X.; LANDIER, A. Why has ceo pay increased so much? *The Quarterly Journal of Economics*, v. 123, n. 1, p. 49–100, 2008. Citado na página 33.

50 TERVIO, M. The difference that ceos make: An assignment model approach. *American Economic Review*, v. 98, n. 3, p. 642–68, 2008. Citado na página 33.

51 RAMOS, J. et al. Using tf-idf to determine word relevance in document queries. In: PISCATAWAY, NJ. *Proceedings of the first instructional conference on machine learning*. [S.l.], 2003. v. 242, p. 133–142. Citado na página 33.

52 SALTON, G.; BUCKLEY, C. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, v. 24, n. 5, p. 513–523, 1988. Citado na página 33.

53 DUMAIS, S. Latent semantic analysis. *Annual Review of Information Science and Technology*, v. 38, n. 1, p. 188–230, 2004. Citado na página 33.

54 HOFMANN, T. Probabilistic latent semantic analysis. In: MORGAN KAUFMANN PUBLISHERS INC. *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*. [S.l.], 1999. p. 289–296. Citado na página 33.

55 BENGIO, Y. et al. Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, Now Publishers Inc., v. 2, n. 1, p. 1–127, 2009. Citado na página 33.

56 HE, X. et al. Neural collaborative filtering. In: INTERNATIONAL WORLD WIDE WEB CONFERENCES STEERING COMMITTEE. *Proceedings of the 26th International Conference on World Wide Web*. [S.l.], 2017. p. 173–182. Citado na página 33.

57 CHE, Y.-K.; KIM, J.; KOJIMA, F. Stable matching in large economies. *Econometrica*, v. 87, n. 1, p. 65–110, 2019. Citado na página 33.

58 SU, X.; KHOSHGOFTAAR, T. M. A survey of collaborative filtering techniques. *Advances in Artificial Intelligence*, Hindawi Publishing Corp., v. 2009, 2009. Citado na página 33.

59 PANG, L. et al. Text matching as image recognition. In: *Thirtieth AAAI Conference on Artificial Intelligence*. [S.l.: s.n.], 2016. Citado na página 33.

60 KOREN, Y.; BELL, R. Advances in collaborative filtering. In: *Recommender Systems Handbook*. [S.l.]: Springer, 2015. p. 77–118. Citado na página 33.

61 BLEI, D.; NG, A.; JORDAN, M. Latent dirichlet allocation. *Journal of Machine Learning Research*, v. 3, n. Jan, p. 993–1022, 2003. Citado na página 33.

62 HE, X.; CHUA, T.-S. Neural factorization machines for sparse predictive analytics. In: ACM. *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. [S.l.], 2017. p. 355–364. Citado na página 33.

63 BAY, H. et al. Speeded-up robust features (surf). *Computer Vision and Image Understanding*, v. 110, n. 3, p. 346–359, 2008. Citado na página 33.

64 GRUEN, A. Adaptive least squares correlation: A powerful image matching technique. *South African Journal of Photogrammetry, Remote Sensing and Cartography*, v. 14, n. 3, p. 175–187, 1985. Citado na página 33.

65 YANG, Q.; WEI, Q. An image matching algorithm based on mutual information for small dimensionality target. In: *2018 14th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*. [S.l.]: IEEE, 2018. p. 1166–1171. Citado na página 33.

66 LOWE, D. G. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, v. 60, n. 2, p. 91–110, 2004. Citado na página 33.

67 ROTH, A. E.; SöNMEZ, T.; ÜNVER, M. U. Kidney exchange: Fair efficient, and incentive-compatible. *American Economic Review*, American Economic Association, v. 95, n. 2, p. 376–380, 2004. Citado na página 33.

68 MANNING, C. D.; RAGHAVAN, P.; SCHüTZE, H. *Introduction to Information Retrieval*. [S.l.]: Cambridge University Press, 2008. Citado na página 34.

69 RICCI, F. et al. *Introduction to Recommender Systems Handbook*. [S.l.]: Springer, 2011. Citado na página 34.

70 JI, S. et al. A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE Transactions on Neural Networks and Learning Systems*, IEEE, v. 33, n. 2, p. 494–514, 2021. Citado na página 34.

71 SZELISKI, R. *Computer Vision: Algorithms and Applications*. [S.l.]: Springer, 2010. Citado na página 34.

72 CHEN, R.; ZHANG, T. Image keypoint matching using graph neural networks. *IEEE Transactions on Image Processing*, 2023. Citado na página 36.

73 GUPTA, A.; SINGH, P. A faster combinatorial algorithm for maximum bipartite matching. *Journal of Graph Theory*, 2020. Citado na página 36.

74 LIU, B.; YANG, F. Energy-delay tradeoff in device-assisted noma mec systems: A matching-based algorithm. *IEEE Transactions on Wireless Communications*, 2022. Citado na página 36.

75 XIAO, L.; WANG, Y. Privacy-preserving graph matching query supporting quick subgraph extraction. *Proceedings of AAAI*, 2021. Citado 2 vezes nas páginas 36 and 37.

76 WANG, S.; LI, D. Graph matching for knowledge graph alignment using edge-coloring propagation. *Knowledge-Based Systems*, 2022. Citado na página 37.

77 REN, J. e. a. Matching algorithms: Fundamentals, applications and challenges. *IEEE Transactions on Emerging Topics in Computational Intelligence*, IEEE, v. 5, n. 3, p. 332–350, 2021. Citado na página 38.

78 LAND, A. H. An automatic algorithm for generating binary search trees. *Computers & Operations Research*, Pergamon, v. 1, n. 1, p. 64–66, 1974. Citado na página 45.

79 FORTIN, F.-A. et al. *DEAP: Evolutionary Algorithms Made Easy*. 2012. <https://deap.readthedocs.io/>. Journal of Machine Learning Research, 13, 2171-2175. Citado 2 vezes nas páginas 45 and 60.

80 HATFIELD, J. W.; MILGROM, P. R. Matching with contracts. *American Economic Review*, JSTOR, v. 95, n. 4, p. 913–935, 2005. Citado 2 vezes nas páginas 47 and 48.

81 ROSTAMI, S.; AL. et. Fairness in kidney exchange programs. In: ACM. *Proceedings of the 2018 ACM Conference on Economics and Computation*. [S.l.], 2018. p. 443–443. Citado na página 47.

82 PAPPALARDO, G.; CONITZER, V. Algorithmic fairness through the prism of fair division. *Artificial Intelligence*, Elsevier, v. 286, p. 103304, 2020. Citado na página 47.

83 MANLOVE, D. *Algorithmics of Matching Under Preferences*. [S.l.]: World Scientific Publishing Company, 2017. Citado 2 vezes nas páginas 47 and 48.

84 YAN, L.; AL. et. Metaheuristic approaches for fair matching in social networks. *Computational Social Networks*, Springer, v. 8, n. 1, p. 1–17, 2021. Citado na página 47.

85 CHAPMAN, A.; AL. et. Matching students to schools with soft and hard constraints. In: IJCAI. *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*. [S.l.], 2017. p. 515–521. Citado na página 47.

86 ROSTAMI, M.; MCELFRESH, D.; DICKERSON, J. P. Matching with diversity constraints: A multi-objective approach. *Proceedings of the AAAI Conference on Artificial Intelligence*, v. 32, n. 1, 2018. Citado na página 48.

87 PAPPALARDO, L.; CONITZER, V. Combining group fairness and efficiency in a machine learning matching market. *Proceedings of the AAAI Conference on Artificial Intelligence*, v. 34, n. 04, p. 5752–5759, 2020. Citado na página 48.

88 YAN, X. et al. An evolutionary approach to fairness and diversity in matching. *Proceedings of the AAAI Conference on Artificial Intelligence*, v. 35, n. 4, p. 3676–3684, 2021. Citado na página 48.

89 CHAPMAN, J. et al. Multi-objective optimization for fair student-supervisor assignments. *Annals of Operations Research*, Springer, v. 252, n. 1, p. 21–40, 2017. Citado na página 48.

90 SANKAR, L. et al. Group fairness for the allocation problem: With application to public housing allocation. In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. [S.l.: s.n.], 2021. Citado na página 48.

91 GARCíA-SORIANO, D.; BONCHI, F. Fair and efficient solutions to the multi-agent resource allocation problem. In: *Proceedings of the 29th ACM International Conference on Information and Knowledge Management*. [S.l.: s.n.], 2020. Citado na página 48.

92 MELLO, R. G. de. *Matching-Optimization-Core*. 2025. Https://github.com/RafaelGranza/Matching-Optimization-Core. Citado na página 57.

93 HAGBERG, A. A.; SCHULT, D. A.; SWART, P. J. *Exploring Network Structure, Dynamics, and Function using NetworkX*. 2008. <https://networkx.org>. Accessed on 29/06/2025. Citado na página 62.

94 CSARDI, G.; NEPUSZ, T. igraph. *International Journal of Complex Systems*, 2006. Disponível em: <https://igraph.org/>. Citado na página 62.

APPENDIX A – GROUPRULE AND GROUP CLASS

```
from collections import defaultdict

class Group:
    """
    A container of members, which can be any type of object.
    Members are organized by their class type.
    """

    def __init__(self):
        self.members = defaultdict(list)

    def add_member(self, *instances):
        if len(instances) == 1 and isinstance(instances[0], (list, set, tuple)):
            instances = instances[0]
        for instance in instances:
            cls = type(instance)
            self.members[cls].append(instance)
        return self

    def remove_member(self, *instances):
        if len(instances) == 1 and isinstance(instances[0], (list, set, tuple)):
            instances = instances[0]
        for instance in instances:
            cls = type(instance)
            if instance in self.members[cls]:
                self.members[cls].remove(instance)
            else:
                raise ValueError(f"Instance {instance} not found in group.")
        return self

    def get_members(self, cls=None):
        if cls is None:
            return self.members
        return self.members.get(cls, [])

    def get_members_as_list(self, cls=None):
        if cls is None:
            return [instance for instances in self.members.values() for instance in
                instances]
        return self.members.get(cls, [])

    def __repr__(self):
        members_repr = (instance for instances in self.members.values() for instance in
            instances)
        return f"Group({', '.join(map(str, members_repr))})"

class GroupRule:
    """
    A class to manage a group of objects with cardinality constraints and validation.
    It allows setting cardinality rules, adding statistics and validators, and defining
        objective functions.
    """

    objective_function = lambda stats: 0
    objective_function_name = "no_statistic"

    """
```

```
    A flag to indicate if the matching should be stable.
    If True, the matching will be stable.
    The instances MUST have a "preference" attribute for this to work.
    """
    stable_match = False

    valid_functions = {
        "minimize_sum_of_single_statistic": lambda stats: sum(stats),
        "minimize_min_of_single_statistic": lambda stats: min(stats),
        "minimize_max_of_single_statistic": lambda stats: max(stats),
        "maximize_sum_of_single_statistic": lambda stats: -sum(stats),
        "maximize_min_of_single_statistic": lambda stats: -min(stats),
        "maximize_max_of_single_statistic": lambda stats: -max(stats),
        "no_statistic": lambda stats: 0, # No statistic, just return 0
    }

    def __init__(self):
        self.cardinality_rules = {}
        self.statistics = []
        self.validators = []
        self.types = []
        self.objective_function_name = None

    def set_objective_function(self, function):
        if isinstance(function, str):
            function_name = function.lower()
            if function_name not in self.valid_functions:
                raise ValueError(f"Invalid objective function name: {function_name}")
            self.objective_function_name = function_name
            self.objective_function = self.valid_functions[function_name]
        else:
            if not callable(function):
                raise ValueError("Objective function must be callable.")
            self.objective_function = function
            self.objective_function_name = "arbitrary"

    def set_cardinality(self, cls, min_count, max_count):
        self.cardinality_rules[cls] = (min_count, max_count)
        if cls not in self.types:
            self.types.append(cls)

    def add_validator(self, validator_fn):
        self.validators.append(validator_fn)

    def add_statistic(self, statistic):
        if not callable(statistic):
            raise ValueError("statistic must be a callable.")
        self.statistics.append(statistic)

    def validate_or_raise(self, group: Group):
        for cls, (min_count, max_count) in self.cardinality_rules.items():
            count = len(group.members[cls])
            if not (min_count <= count <= max_count):
                raise ValueError(f"Cardinality constraint violated for class {cls.
                    __name__}: found {count}, expected from {min_count} to {max_count}.")

        for validator in self.validators:
            if not validator(group.members):
                raise ValueError(f"Custom validator '{validator.__name__}' failed.")
```

```
    def validate(self, groups: Group):
        try:
            self.validate_or_raise(groups)
        except ValueError as e:
            return False
        return True

    def set_stable_match(self, stable_match: bool):
        self.stable_match = stable_match
```

Script A.1 – GroupRule and Group Class Source Code

APPENDIX B – HUNGARIAN ALGORITHM SOLVER

```python
from src.solvers.base_solver import Solver
from src.group import Group, GroupRule
from typing import List, Type
import numpy as np
from scipy.optimize import linear_sum_assignment


def build_cost_matrix_from_groups(group_rule, groups):
    """
    Build a cost matrix from the given groups, considering two types of objects.
    """
    # count the unique instances of each type in the groups
    unique_instances = {}
    for group in groups:
        for cls, instance   in group.members.items():
            if cls not in unique_instances:
                unique_instances[cls] = list()
            for i in instance:
                if i not in unique_instances[cls]:
                    unique_instances[cls].append(i)

    [type_a, type_b] = unique_instances.values()
    n = len(type_a)
    m = len(type_b)
    cost_matrix = np.zeros((n, m))

    for i in range(n):
        for j in range(m):
            group = Group()
            group.add_member(unique_instances[group_rule.types[0]][i])
            group.add_member(unique_instances[group_rule.types[1]][j])
            cost_matrix[i][j] = group_rule.statistics[0](group.members)

    return cost_matrix

def recover_groups_from_groups(matching, groups):
    """
    Recover the groups from the matching result.
    """
    unique_instances = {}
    for group in groups:
        for cls, instance   in group.members.items():
            if cls not in unique_instances:
                unique_instances[cls] = list()
            for i in instance:
                if i not in unique_instances[cls]:
                    unique_instances[cls].append(i)

    types = list(unique_instances.keys())

    result_groups = []
    for i, j in matching:
        group = Group()
        group.add_member(unique_instances[types[0]][i])
        group.add_member(unique_instances[types[1]][j])
        result_groups.append(group)
    return result_groups
```

```
def build_cost_matrix(group_rule, type_a, type_b):
    """
    Build a cost matrix from the given instances, considering two types of objects.
    """
    n, m = len(type_a), len(type_b)
    cost_matrix = np.zeros((n, m))

    for i in range(n):
        for j in range(m):
            group = Group()
            group.add_member(type_a[i])
            group.add_member(type_b[j])
            cost_matrix[i][j] = group_rule.statistics[0](group.members)

    return cost_matrix

def recover_groups(matching, type_a, type_b):
    """
    Recover the groups from the matching result.
    """
    groups = []
    for i, j in matching:
        group = Group()
        group.add_member(type_a[i])
        group.add_member(type_b[j])
        groups.append(group)
    return groups

class HungarianAlgorithm(Solver):
    """
    Solver using the Hungarian algorithm for assignment problems.
    """

    @staticmethod
    def can_solve(group_rule: GroupRule):
        if len(group_rule.cardinality_rules) != 2:
            return False
        for cls, (min_count, max_count) in group_rule.cardinality_rules.items():
            if min_count != 1 or max_count != 1:
                return False
        return group_rule.objective_function_name == "minimize_sum_of_single_statistic"

    @staticmethod
    def solve_from_instances(group_rule: GroupRule, instances: List[object]):

        type_a = [inst for inst in instances if isinstance(inst, group_rule.types[0])]
        type_b = [inst for inst in instances if isinstance(inst, group_rule.types[1])]

        matrix = build_cost_matrix(group_rule, type_a, type_b)
        row_ind, col_ind = linear_sum_assignment(matrix)
        matching = list(zip(row_ind, col_ind))
        return recover_groups(matching, type_a, type_b)

    @staticmethod
    def solve_from_valid_groups(group_rule: GroupRule, groups: List[Group]):
        matrix = build_cost_matrix_from_groups(group_rule, groups)
        row_ind, col_ind = linear_sum_assignment(matrix)
```

```
        matching = list(zip(row_ind, col_ind))
        return recover_groups_from_groups(matching, groups)
```

Script B.1 – Hungarian Algorithm Solver Source Code

APPENDIX C – HUNGARIAN ALGORITHM WITH BINARY SEARCH SOLVER

```python
from src.solvers.base_solver import Solver
from src.group import Group, GroupRule
from typing import List, Type
import numpy as np
from scipy.optimize import linear_sum_assignment


def build_cost_matrix_from_groups(group_rule, groups):
    """
    Build a cost matrix from the given groups, considering two types of objects.
    """
    # count the unique instances of each type in the groups
    unique_instances = {}
    for group in groups:
        for cls, instance   in group.members.items():
            if cls not in unique_instances:
                unique_instances[cls] = list()
            for i in instance:
                if i not in unique_instances[cls]:
                    unique_instances[cls].append(i)

    [type_a, type_b] = unique_instances.values()
    n = len(type_a)
    m = len(type_b)
    cost_matrix = np.zeros((n, m))

    for i in range(n):
        for j in range(m):
            group = Group()
            group.add_member(unique_instances[group_rule.types[0]][i])
            group.add_member(unique_instances[group_rule.types[1]][j])
            cost_matrix[i][j] = group_rule.statistics[0](group.members)

    return cost_matrix

def recover_groups_from_groups(matching, groups):
    """
    Recover the groups from the matching result.
    """
    unique_instances = {}
    for group in groups:
        for cls, instance   in group.members.items():
            if cls not in unique_instances:
                unique_instances[cls] = list()
            for i in instance:
                if i not in unique_instances[cls]:
                    unique_instances[cls].append(i)

    types = list(unique_instances.keys())

    result_groups = []
    for i, j in matching:
        group = Group()
        group.add_member(unique_instances[types[0]][i])
        group.add_member(unique_instances[types[1]][j])
        result_groups.append(group)
    return result_groups
```

```
def bin_search(matrix):
    matching = []
    l = 0
    r = max(all_costs := matrix.flatten())
    iters = 100

    while iters:
        mid = (l + r) / 2
        aux_matrix = np.where(matrix > mid, np.inf, matrix)

        row_ind, col_ind = linear_sum_assignment(matrix)
        matching = list(zip(row_ind, col_ind))
        if matrix[row_ind, col_ind].sum() >= np.inf:
            r = mid
        else:
            l = mid
        iters -= 1
    return matching

def build_cost_matrix(group_rule, type_a, type_b):
    """
    Build a cost matrix from the given instances, considering two types of objects.
    """
    n, m = len(type_a), len(type_b)
    cost_matrix = np.zeros((n, m))

    for i in range(n):
        for j in range(m):
            group = Group()
            group.add_member(type_a[i])
            group.add_member(type_b[j])
            cost_matrix[i][j] = group_rule.statistics[0](group.members)

    return cost_matrix

def recover_groups(matching, type_a, type_b):
    """
    Recover the groups from the matching result.
    """
    groups = []
    for i, j in matching:
        group = Group()
        group.add_member(type_a[i])
        group.add_member(type_b[j])
        groups.append(group)
    return groups

class BinSearchHungarianAlgorithm(Solver):
    """
    Solver using a binary search approach combined with the Hungarian algorithm.
    This solver is specifically designed for the case where there are exactly two types
        of objects
    and each type has a cardinality of 1.
    It finds the Minumum individual cost for each group and returns the groups.
    """

    @staticmethod
```

```
    def can_solve(group_rule: GroupRule):
        if len(group_rule.cardinality_rules) != 2:
            return False
        for cls, (min_count, max_count) in group_rule.cardinality_rules.items():
            if min_count != 1 or max_count != 1:
                return False
        return group_rule.objective_function_name == "minimize_max_of_single_statistic"

    @staticmethod
    def solve_from_instances(group_rule: GroupRule, instances: List[object]):

        type_a = [inst for inst in instances if isinstance(inst, group_rule.types[0])]
        type_b = [inst for inst in instances if isinstance(inst, group_rule.types[1])]

        matrix = build_cost_matrix(group_rule, type_a, type_b)
        matching = bin_search(matrix)
        return recover_groups(matching, type_a, type_b)

    @staticmethod
    def solve_from_valid_groups(group_rule: GroupRule, groups: List[Group]):
        matrix = build_cost_matrix_from_groups(group_rule, groups)
        matching = bin_search(matrix)
        return recover_groups_from_groups(matching, groups)
```

Script C.1 – Hungarian Algorithm with Binary Search Solver Source Code

APPENDIX D – STABLE MARRIAGE SOLVER

```
from src.solvers.base_solver import Solver
from src.group import Group, GroupRule
from typing import List, Type


def check_preference(prefer, n, m0, m1):
    """
    Check if element N prefers element M0 over element M1.
    """
    return prefer[wn.index(m1)] < prefer[n].index(m0)


def make_stable_pairs(type_a, type_b):
    """
    Stable Marriage algorithm to find stable pairs between two types.
    """
    # Assume type_a and type_b have a 'preference' attribute that is a list of
        preferences
    for a in type_a:
        if not hasattr(a, 'preference'):
            raise ValueError(f"Instances of type {type(a).__name__} must have a '
                preference' attribute.")
    for b in type_b:
        if not hasattr(b, 'preference'):
            raise ValueError(f"Instances of type {type(b).__name__} must have a '
                preference' attribute.")

    # Build preference lists
    prefer = {a: a.preference for a in type_a}
    prefer.update({b: b.preference for b in type_b})

    w_partner = {b: None for b in type_b}  # Women's partners
    m_free = set(type_a)  # Free men

    while m_free:
        m = m_free.pop()
        for w in prefer[m]:
            if w_partner[w] is None:  # Woman is free
                w_partner[w] = m
                break
            else:  # Woman is already engaged
                m1 = w_partner[w]
                if not check_preference(prefer, w, m, m1):
                    w_partner[w] = m
                    m_free.add(m1)
                    break

    return [(m, w) for w, m in w_partner.items()]

def build_groups(stable_pairs):
    """
    Build groups from stable pairs.
    """
    groups = []
    for m, w in stable_pairs:
        group = Group()
        group.add_member(m)
```

```
            group.add_member(w)
            groups.append(group)
        return groups


class StableMarriage(Solver):
    """
    Solver using the Stable Marriage algorithm for assignment problems.
    """

    @staticmethod
    def can_solve(group_rule: GroupRule):
        """
        Check if the group rule can be solved using the Stable Marriage algorithm.
        """
        if len(group_rule.cardinality_rules) != 2:
            return False
        for _, (min_count, max_count) in group_rule.cardinality_rules.items():
            if min_count != 1 or max_count != 1:
                return False
        return group_rule.stable_match and group_rule.objective_function_name == "
            no_statistic"

    @staticmethod
    def solve_from_instances(group_rule: GroupRule, instances: List[object]):
        """
        Solve the problem using the Stable Marriage algorithm.
        """
        type_a = [inst for inst in instances if isinstance(inst, group_rule.types[0])]
        type_b = [inst for inst in instances if isinstance(inst, group_rule.types[1])]

        stable_pairs = make_stable_pairs(type_a, type_b)
        return build_groups(stable_pairs)

    @staticmethod
    def solve_from_valid_groups(group_rule: GroupRule, groups: List[Group]):

        unique_instances = {}
        for group in groups:
            for cls, instance in group.members.items():
                if cls not in unique_instances:
                    unique_instances[cls] = list()
                for i in instance:
                    if i not in unique_instances[cls]:
                        unique_instances[cls].append(i)

        [type_a, type_b] = unique_instances.values()

        stable_pairs = make_stable_pairs(type_a, type_b)
        return build_groups(stable_pairs)
```

Script D.1 – Stable Marriage Solver Source Code

APPENDIX E – GENETIC ALGORITHM SOLVER

```
from src.solvers.base_solver import Solver
from src.group import Group, GroupRule
from typing import List, Type
import numpy as np


def optimize_by_instances(instances: List[object], group_rule: GroupRule, n_groups=3,
    n_generations=10000, pop_size=40, mutation_prob=0.5) -> List[Group]:
    '''
    The chromosome is a binary vector of size n_groups * n_instances, representing a
        n_groups x n_instances matrix.
    An element can belong to more than one group.
    '''
    from deap import base, creator, tools, algorithms
    import random
    import numpy as np

    m = len(instances)
    n = n_groups

    creator.create("FitnessMax", base.Fitness, weights=(2.0,))
    creator.create("Individual", list, fitness=creator.FitnessMax)

    toolbox = base.Toolbox()

    def random_individual():
        # Binary vector of size n*m
        return [random.randint(0, 1) for _ in range(n * m)]

    toolbox.register("individual", tools.initIterate, creator.Individual,
        random_individual)
    toolbox.register("population", tools.initRepeat, list, toolbox.individual)

    def mate(parent1, parent2):
        # Uniform crossover per gene
        child1 = [parent1[i] if random.random() < 0.5 else parent2[i] for i in range(n *
            m)]
        child2 = [parent2[i] if random.random() < 0.5 else parent1[i] for i in range(n *
            m)]
        return creator.Individual(child1), creator.Individual(child2)

    def mutate(ind):
        for i in range(n * m):
            if random.random() < mutation_prob:
                ind[i] = 1 - ind[i]
        return (ind,)

    def eval_grouping(ind):
        # Converts vector to n x m matrix
        mat = np.array(ind).reshape((n, m))
        group_objs = []
        for i in range(n):
            members = [instances[j] for j in range(m) if mat[i, j] == 1]
            if members:
                g = Group()
                g.add_member(members)
                group_objs.append(g)
```

```
        score = 0
        # Validation
        for g in group_objs:
            if not group_rule.validate(g):
                score += -1e8
        # Statistics
        score += group_rule.objective_function(group_objs) if group_objs else 0
        return (score,)

    toolbox.register("evaluate", eval_grouping)
    toolbox.register("mate", mate)
    toolbox.register("mutate", mutate)
    toolbox.register("select", tools.selTournament, tournsize=3)

    pop = toolbox.population(n=pop_size)
    hof = tools.HallOfFame(1)

    algorithms.eaSimple(pop, toolbox, cxpb=0.7, mutpb=0.3, ngen=n_generations, halloffame
        =hof, verbose=False)

    best = hof[0]
    mat = np.array(best).reshape((n, m))
    group_objs = []
    for i in range(n):
        members = [instances[j] for j in range(m) if mat[i, j] == 1]
        if members:
            g = Group()
            g.add_member(members)
            group_objs.append(g)
    return group_objs

def optimize_by_groups(groups: List[Group], group_rule: GroupRule, n_generations=10000,
    pop_size=30, mutation_prob=0.2) -> List[Group]:
    """
    The chromosome is a bit mask b of size n, where n is the number of
    allowed Grouping Objects. The i-th bit in b indicates whether or not the i-th
        grouping object is
    selected at that solution option.
    """
    from deap import base, creator, tools, algorithms
    import random

    n = len(groups)

    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
    creator.create("Individual", list, fitness=creator.FitnessMax)

    toolbox = base.Toolbox()
    toolbox.register("attr_bool", random.randint, 0, 1)
    toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.
        attr_bool, n)
    toolbox.register("population", tools.initRepeat, list, toolbox.individual)

    def eval_selection(individual):
        selected_groups = [g for i, g in enumerate(groups) if individual[i]]
        score = 0
        # Statistics
        score = group_rule.objective_function(selected_groups) if selected_groups else 0
        return (score,)
```

```
    toolbox.register("evaluate", eval_selection)
    toolbox.register("mate", tools.cxTwoPoint)
    toolbox.register("mutate", tools.mutFlipBit, indpb=mutation_prob)
    toolbox.register("select", tools.selTournament, tournsize=3)

    pop = toolbox.population(n=pop_size)
    hof = tools.HallOfFame(1)

    algorithms.eaSimple(pop, toolbox, cxpb=0.7, mutpb=0.3, ngen=n_generations, halloffame
        =hof, verbose=False)

    best = hof[0]
    selected_groups = [g for i, g in enumerate(groups) if best[i]]
    return selected_groups

class GeneticAlgorithm(Solver):

    @staticmethod
    def can_solve(group_rule: GroupRule):
        return True

    @staticmethod
    def solve_from_instances(group_rule: GroupRule, instances: List[object]):
        answers = optimize_by_instances(instances, group_rule, n_groups=len(instances))
        return answers

    def solve_from_valid_groups(group_rule: GroupRule, groups: List[Group]):
        answers = optimize_by_groups(groups, group_rule)
        return answers
```

Script E.1 – Genetic Algorithm Solver Source Code

APPENDIX F – FAIR BIPARTITE MATCHING SOLVER

```
\begin{lstlisting}
from src.solvers.base_solver import Solver
from src.group import Group, GroupRule
from typing import List, Type

import igraph as ig
import matplotlib.pyplot as plt
import numpy as np
import networkx as nx
import random as rd

class quota:
    def __init__(self, characteristic, distribution, scale=True):
        self.characteristic = characteristic
        self.distribution = distribution
        total = sum([ x[1] for x in self.distribution ])
        if scale:
            map(lambda x: [x[0], x[1]/total] , self.distribution)
        elif total < 1:
            distribution.append(['Remaining', 1-total])

    def __str__(self):
        return "{ " + str(self.characteristic) + ": " + str(self.distribution) + " }"

    def __mul__(self, obj):
        if len(self.distribution) == 0:
            return obj

        if len(obj.distribution) == 0:
            return self

        new_distribution = []
        for distribution_1 in self.distribution:
            for distribution_2 in obj.distribution:
                new_distribution.append([distribution_1[0] + '+' + distribution_2[0],
                    distribution_1[1] * distribution_2[1]])
        return quota(self.characteristic + '/' + obj.characteristic, new_distribution)

class quotas_description:
    def __init__(self, quotas):
        self.requirement = self.combine_quotas(quotas)

    def __str__(self):
        return str(self.requirement)

    def characteristic(self):
        return self.requirement.characteristic

    def distribution(self):
        return self.requirement.distribution

    def combine_quotas(self, quotas):
        new_requirement = quota("", [])
        for requirement in quotas:
            new_requirement *= requirement
        return new_requirement
proxy={}
```

```
class mapper:
    def __init__(self, group_a, group_b, matches, quotas_group_a):
        self.group_a = group_a
        self.group_b = group_b
        self.matches = matches
        self.quotas_group_a = quotas_group_a
        self.number_of_matches = len(group_b)

        self.graph = self.build_graph()

    def build_graph(self):
        global proxy
        proxy={}
        g = ig.Graph(directed=True)
        G = nx.Graph()
        G.add_node(len(g.vs), subset=0, name="source", label='Source')
        g.add_vertex(type='source', name="source")


        remaining_matches = self.number_of_matches
        for group in self.quotas_group_a.requirement.distribution:
            G.add_node(len(g.vs), name=group[0], label='Fair Group', quotas=group[1],
                subset=1, obj=group)
            g.add_vertex(name=group[0], quotas=group[1], type='Fair Group', obj=group)
            G.add_edge(0, len(g.vs)-1, capacity=group[1], weight=0)
            g.add_edge(0, len(g.vs)-1, capacity=group[1], weight=0)
            remaining_matches-=group[1]

        # Remaining
        if len(self.quotas_group_a.requirement.distribution):
            G.add_node(len(g.vs), name="Remaining", label='Fair Group', subset=1, obj={'
                extra': True})
            g.add_vertex(name="Remaining", type='Fair Group', obj={'extra': True})
            G.add_edge(0, len(g.vs)-1, capacity=remaining_matches, weight=0)
            g.add_edge(0, len(g.vs)-1, capacity=remaining_matches, weight=0)

        for obj in self.group_a:
            G.add_node(len(g.vs), obj=obj, subset=3, name="Worker", label='Worker')
            g.add_vertex(obj=obj, type='group_a')

        self.add_edges_group_and_requirement(self.group_a, self.quotas_group_a, 0, g, G)


        for obj in self.group_b:
            G.add_node(len(g.vs), obj=obj, subset=4, name="Job", label='Job')
            g.add_vertex(obj=obj, type='group_b')


        for hash in self.matches:
            [[_, u], [_, v], [_, w]] = hash.items()
            G.add_edge(g.vs.find(obj=self.group_a[u]).index, g.vs.find(obj=self.group_b[v
                ]).index, capacity=1, weight=w)
            g.add_edge(g.vs.find(obj=self.group_a[u]).index, g.vs.find(obj=self.group_b[v
                ]).index, capacity=1, weight=w)

        G.add_node(len(g.vs), subset=6, name="target", label='Target')
        g.add_vertex(type='target', name="target")
```

```
        self.add_edges_group_and_requirement(self.group_b, quotas_description([]), len(g.
            vs)-1, g, G)


        return G

    def has_quotas(self, quotas):
        return len(quotas.requirement.distribution) >= 1

    def add_edges_group_and_requirement(self, group, requirement, in_case_its_empty, g, G
        ):
        list_of_characteristics = requirement.requirement.characteristic.split('/')
        if len(requirement.requirement.distribution) == 0:
            for obj in group:
                g.add_edge(in_case_its_empty, g.vs.find(obj=obj).index, capacity=1,
                    weight=0)
                G.add_edge(in_case_its_empty, g.vs.find(obj=obj).index, capacity=1,
                    weight=0)

        else:


            list_proxy=[]
            for dist in requirement.requirement.distribution:
                list_of_distribution = dist[0].split('/')
                for obj in group:
                    if not any([getattr(obj, list_of_characteristics[i], None) !=
                        list_of_distribution[i] for i in range(len(list_of_distribution))
                        ]):
                        if obj not in proxy.values():
                            key = rd.random()
                            G.add_node(len(g.vs), subset=2, obj=key, name="proxy", label=
                                'proxy')
                            g.add_vertex(name='proxy', type='proxy', obj=key)
                            g.add_edge(len(g.vs)-1, g.vs.find(obj=obj).index, capacity=1,
                                 weight=0, type='proxy')
                            G.add_edge(len(g.vs)-1, g.vs.find(obj=obj).index, capacity=1,
                                 weight=0, type='proxy')
                            proxy[key] = obj
                        g.add_edge(g.vs.find(obj=dist).index, len(g.vs)-1, capacity=1,
                            weight=0)
                        G.add_edge(g.vs.find(obj=dist).index, len(g.vs)-1, capacity=1,
                            weight=0)



            for obj in group:
                if obj not in proxy.values():
                    key=rd.random()
                    G.add_node(len(g.vs), subset=2, obj=key, name="proxy", label='proxy')
                    g.add_vertex(name='proxy', type='proxy', obj=key)
                    g.add_edge(len(g.vs)-1, g.vs.find(obj=obj).index, capacity=1, weight
                        =0, type='proxy')
                    G.add_edge(len(g.vs)-1, g.vs.find(obj=obj).index, capacity=1, weight
                        =0, type='proxy')
                    proxy[key] = obj
                proxy_obj = [i for i in proxy if proxy[i]==obj][0]
```

```
                g.add_edge(g.vs.find(obj={'extra': True}).index, g.vs.find(obj=proxy_obj)
                    .index, capacity=1, weight=0)
                G.add_edge(g.vs.find(obj={'extra': True}).index, g.vs.find(obj=proxy_obj)
                    .index, capacity=1, weight=0)

def update_graph(G, mincostFlow):
    for u in mincostFlow:
        for v in mincostFlow[u]:
            if u>v and G.edges[u, v].get('type') == 'proxy' and  G.edges[u, v]["used"
                ]==0:
                 G.edges[u, v]["used"] = mincostFlow[u][v]
            if u > v: continue
            G.edges[u, v]["used"] = mincostFlow[u][v]

def solve(G):
    mincostFlow = nx.max_flow_min_cost(G, 0, len(G.nodes)-1)
    update_graph(G, mincostFlow)
    return[nx.maximum_flow_value(G, 0, len(G.nodes)-1), nx.cost_of_flow(G, mincostFlow)]


def gen_matching_from_instances(gr, type_a, type_b):
    list = []
    for source in range(len(type_a)):
        for destiny in range(len(type_b)):
            group = Group()
            group.add_member(type_a[source])
            group.add_member(type_b[destiny])
            list.append({
                "source": source,
                "destiny": destiny,
                'weight': gr.statistics[0](group.members)
            })
    return list

def gen_matching_from_groups(gr, groups):
    list = []
    for group in groups:
        for a in group.members[0]:
            for b in group.members[1]:
                list.append({
                    "source": a.ID,
                    "destiny": b.ID,
                    'weight': gr.statistics[0](group.members)
                })
    return list

def build_groups(m):
    used_edges = [(u, v) for u, v, d in m.graph.edges(data=True) if 'used' in d and d['
        used'] > 0]
    global proxy
    fair_pairs = []
    for u, v in used_edges:
        if m.graph.nodes[u]['label'] == 'Worker' and m.graph.nodes[v]['label'] == 'Job':
            fair_pairs.append((m.graph.nodes[u]['obj'], m.graph.nodes[v]['obj']))

    groups = []
    for a, b in fair_pairs:
        group = Group()
        group.add_member(a)
```

```
        group.add_member(b)
        groups.append(group)
    return groups

class FairBipartiteMatching(Solver):
    """
    Solver using the Fair Bipartite algorithm for assignment problems.
    """

    @staticmethod
    def can_solve(group_rule: GroupRule):
        if len(group_rule.cardinality_rules) != 2:
            return False
        for _, (min_count, max_count) in group_rule.cardinality_rules.items():
            if min_count != 1 or max_count != 1:
                return False

        # check if there is a validator called "fair_matching"
        return len(group_rule.quotas) > 0

    @staticmethod
    def solve_from_instances(group_rule: GroupRule, instances: List[object]):

        quotas = quotas_description([quota(quota_name, quota_definition) for quota_name,
            quota_definition in group_rule.quotas.items()])

        type_a = [inst for inst in instances if isinstance(inst, group_rule.types[0])]
        type_b = [inst for inst in instances if isinstance(inst, group_rule.types[1])]

        matching = gen_matching_from_instances(group_rule, type_a, type_b)
        m = mapper(type_a, type_b, matching, quotas)
        solve(m.graph)

        return build_groups(m)


    @staticmethod
    def solve_from_valid_groups(group_rule: GroupRule, groups: List[Group]):
        unique_instances = {}
        for group in groups:
            for cls, instance in group.members.items():
                if cls not in unique_instances:
                    unique_instances[cls] = list()
                for i in instance:
                    if i not in unique_instances[cls]:
                        unique_instances[cls].append(i)

        [type_a, type_b] = unique_instances.values()

        matching = gen_matching_from_groups(group_rule, groups)

        m = mapper(type_a, type_b, matching, quotas)
        solve(m.graph)

        return build_groups(m)
```

Script F.1 – Fair Bipartite Matching Solver Source Code

APPENDIX A – GROUPRULE AND GROUP CLASS

```
from collections import defaultdict

class Group:
    """
    A container of members, which can be any type of object.
    Members are organized by their class type.
    """

    def __init__(self):
        self.members = defaultdict(list)

    def add_member(self, *instances):
        if len(instances) == 1 and isinstance(instances[0], (list, set, tuple)):
            instances = instances[0]
        for instance in instances:
            cls = type(instance)
            self.members[cls].append(instance)
        return self

    def remove_member(self, *instances):
        if len(instances) == 1 and isinstance(instances[0], (list, set, tuple)):
            instances = instances[0]
        for instance in instances:
            cls = type(instance)
            if instance in self.members[cls]:
                self.members[cls].remove(instance)
            else:
                raise ValueError(f"Instance {instance} not found in group.")
        return self

    def get_members(self, cls=None):
        if cls is None:
            return self.members
        return self.members.get(cls, [])

    def get_members_as_list(self, cls=None):
        if cls is None:
            return [instance for instances in self.members.values() for instance in
                instances]
        return self.members.get(cls, [])

    def __repr__(self):
        members_repr = (instance for instances in self.members.values() for instance in
            instances)
        return f"Group({', '.join(map(str, members_repr))})"

class GroupRule:
    """
    A class to manage a group of objects with cardinality constraints and validation.
    It allows setting cardinality rules, adding statistics and validators, and defining
        objective functions.
    """

    objective_function = lambda stats: 0
    objective_function_name = "no_statistic"

    """
```

```
    A flag to indicate if the matching should be stable.
    If True, the matching will be stable.
    The instances MUST have a "preference" attribute for this to work.
    """
    stable_match = False

    valid_functions = {
        "minimize_sum_of_single_statistic": lambda stats: sum(stats),
        "minimize_min_of_single_statistic": lambda stats: min(stats),
        "minimize_max_of_single_statistic": lambda stats: max(stats),
        "maximize_sum_of_single_statistic": lambda stats: -sum(stats),
        "maximize_min_of_single_statistic": lambda stats: -min(stats),
        "maximize_max_of_single_statistic": lambda stats: -max(stats),
        "no_statistic": lambda stats: 0, # No statistic, just return 0
    }

    def __init__(self):
        self.cardinality_rules = {}
        self.statistics = []
        self.validators = []
        self.types = []
        self.objective_function_name = None

    def set_objective_function(self, function):
        if isinstance(function, str):
            function_name = function.lower()
            if function_name not in self.valid_functions:
                raise ValueError(f"Invalid objective function name: {function_name}")
            self.objective_function_name = function_name
            self.objective_function = self.valid_functions[function_name]
        else:
            if not callable(function):
                raise ValueError("Objective function must be callable.")
            self.objective_function = function
            self.objective_function_name = "arbitrary"

    def set_cardinality(self, cls, min_count, max_count):
        self.cardinality_rules[cls] = (min_count, max_count)
        if cls not in self.types:
            self.types.append(cls)

    def add_validator(self, validator_fn):
        self.validators.append(validator_fn)

    def add_statistic(self, statistic):
        if not callable(statistic):
            raise ValueError("statistic must be a callable.")
        self.statistics.append(statistic)

    def validate_or_raise(self, group: Group):
        for cls, (min_count, max_count) in self.cardinality_rules.items():
            count = len(group.members[cls])
            if not (min_count <= count <= max_count):
                raise ValueError(f"Cardinality constraint violated for class {cls.
                    __name__}: found {count}, expected from {min_count} to {max_count}.")

        for validator in self.validators:
            if not validator(group.members):
                raise ValueError(f"Custom validator '{validator.__name__}' failed.")
```

```
    def validate(self, groups: Group):
        try:
            self.validate_or_raise(groups)
        except ValueError as e:
            return False
        return True

    def set_stable_match(self, stable_match: bool):
        self.stable_match = stable_match
```

Script A.1 – GroupRule and Group Class Source Code

APPENDIX B – HUNGARIAN ALGORITHM SOLVER

```python
from src.solvers.base_solver import Solver
from src.group import Group, GroupRule
from typing import List, Type
import numpy as np
from scipy.optimize import linear_sum_assignment


def build_cost_matrix_from_groups(group_rule, groups):
    """
    Build a cost matrix from the given groups, considering two types of objects.
    """
    # count the unique instances of each type in the groups
    unique_instances = {}
    for group in groups:
        for cls, instance   in group.members.items():
            if cls not in unique_instances:
                unique_instances[cls] = list()
            for i in instance:
                if i not in unique_instances[cls]:
                    unique_instances[cls].append(i)

    [type_a, type_b] = unique_instances.values()
    n = len(type_a)
    m = len(type_b)
    cost_matrix = np.zeros((n, m))

    for i in range(n):
        for j in range(m):
            group = Group()
            group.add_member(unique_instances[group_rule.types[0]][i])
            group.add_member(unique_instances[group_rule.types[1]][j])
            cost_matrix[i][j] = group_rule.statistics[0](group.members)

    return cost_matrix

def recover_groups_from_groups(matching, groups):
    """
    Recover the groups from the matching result.
    """
    unique_instances = {}
    for group in groups:
        for cls, instance   in group.members.items():
            if cls not in unique_instances:
                unique_instances[cls] = list()
            for i in instance:
                if i not in unique_instances[cls]:
                    unique_instances[cls].append(i)

    types = list(unique_instances.keys())

    result_groups = []
    for i, j in matching:
        group = Group()
        group.add_member(unique_instances[types[0]][i])
        group.add_member(unique_instances[types[1]][j])
        result_groups.append(group)
    return result_groups
```

```
def build_cost_matrix(group_rule, type_a, type_b):
    """
    Build a cost matrix from the given instances, considering two types of objects.
    """
    n, m = len(type_a), len(type_b)
    cost_matrix = np.zeros((n, m))

    for i in range(n):
        for j in range(m):
            group = Group()
            group.add_member(type_a[i])
            group.add_member(type_b[j])
            cost_matrix[i][j] = group_rule.statistics[0](group.members)

    return cost_matrix

def recover_groups(matching, type_a, type_b):
    """
    Recover the groups from the matching result.
    """
    groups = []
    for i, j in matching:
        group = Group()
        group.add_member(type_a[i])
        group.add_member(type_b[j])
        groups.append(group)
    return groups

class HungarianAlgorithm(Solver):
    """
    Solver using the Hungarian algorithm for assignment problems.
    """

    @staticmethod
    def can_solve(group_rule: GroupRule):
        if len(group_rule.cardinality_rules) != 2:
            return False
        for cls, (min_count, max_count) in group_rule.cardinality_rules.items():
            if min_count != 1 or max_count != 1:
                return False
        return group_rule.objective_function_name == "minimize_sum_of_single_statistic"

    @staticmethod
    def solve_from_instances(group_rule: GroupRule, instances: List[object]):

        type_a = [inst for inst in instances if isinstance(inst, group_rule.types[0])]
        type_b = [inst for inst in instances if isinstance(inst, group_rule.types[1])]

        matrix = build_cost_matrix(group_rule, type_a, type_b)
        row_ind, col_ind = linear_sum_assignment(matrix)
        matching = list(zip(row_ind, col_ind))
        return recover_groups(matching, type_a, type_b)

    @staticmethod
    def solve_from_valid_groups(group_rule: GroupRule, groups: List[Group]):
        matrix = build_cost_matrix_from_groups(group_rule, groups)
        row_ind, col_ind = linear_sum_assignment(matrix)
```

```
        matching = list(zip(row_ind, col_ind))
        return recover_groups_from_groups(matching, groups)
```

Script B.1 – Hungarian Algorithm Solver Source Code

APPENDIX C – HUNGARIAN ALGORITHM WITH BINARY SEARCH SOLVER

```python
from src.solvers.base_solver import Solver
from src.group import Group, GroupRule
from typing import List, Type
import numpy as np
from scipy.optimize import linear_sum_assignment


def build_cost_matrix_from_groups(group_rule, groups):
    """
    Build a cost matrix from the given groups, considering two types of objects.
    """
    # count the unique instances of each type in the groups
    unique_instances = {}
    for group in groups:
        for cls, instance   in group.members.items():
            if cls not in unique_instances:
                unique_instances[cls] = list()
            for i in instance:
                if i not in unique_instances[cls]:
                    unique_instances[cls].append(i)

    [type_a, type_b] = unique_instances.values()
    n = len(type_a)
    m = len(type_b)
    cost_matrix = np.zeros((n, m))

    for i in range(n):
        for j in range(m):
            group = Group()
            group.add_member(unique_instances[group_rule.types[0]][i])
            group.add_member(unique_instances[group_rule.types[1]][j])
            cost_matrix[i][j] = group_rule.statistics[0](group.members)

    return cost_matrix

def recover_groups_from_groups(matching, groups):
    """
    Recover the groups from the matching result.
    """
    unique_instances = {}
    for group in groups:
        for cls, instance   in group.members.items():
            if cls not in unique_instances:
                unique_instances[cls] = list()
            for i in instance:
                if i not in unique_instances[cls]:
                    unique_instances[cls].append(i)

    types = list(unique_instances.keys())

    result_groups = []
    for i, j in matching:
        group = Group()
        group.add_member(unique_instances[types[0]][i])
        group.add_member(unique_instances[types[1]][j])
        result_groups.append(group)
    return result_groups
```

```
def bin_search(matrix):
    matching = []
    l = 0
    r = max(all_costs := matrix.flatten())
    iters = 100

    while iters:
        mid = (l + r) / 2
        aux_matrix = np.where(matrix > mid, np.inf, matrix)

        row_ind, col_ind = linear_sum_assignment(matrix)
        matching = list(zip(row_ind, col_ind))
        if matrix[row_ind, col_ind].sum() >= np.inf:
            r = mid
        else:
            l = mid
        iters -= 1
    return matching

def build_cost_matrix(group_rule, type_a, type_b):
    """
    Build a cost matrix from the given instances, considering two types of objects.
    """
    n, m = len(type_a), len(type_b)
    cost_matrix = np.zeros((n, m))

    for i in range(n):
        for j in range(m):
            group = Group()
            group.add_member(type_a[i])
            group.add_member(type_b[j])
            cost_matrix[i][j] = group_rule.statistics[0](group.members)

    return cost_matrix

def recover_groups(matching, type_a, type_b):
    """
    Recover the groups from the matching result.
    """
    groups = []
    for i, j in matching:
        group = Group()
        group.add_member(type_a[i])
        group.add_member(type_b[j])
        groups.append(group)
    return groups

class BinSearchHungarianAlgorithm(Solver):
    """
    Solver using a binary search approach combined with the Hungarian algorithm.
    This solver is specifically designed for the case where there are exactly two types
        of objects
    and each type has a cardinality of 1.
    It finds the Minumum individual cost for each group and returns the groups.
    """

    @staticmethod
```

```
    def can_solve(group_rule: GroupRule):
        if len(group_rule.cardinality_rules) != 2:
            return False
        for cls, (min_count, max_count) in group_rule.cardinality_rules.items():
            if min_count != 1 or max_count != 1:
                return False
        return group_rule.objective_function_name == "minimize_max_of_single_statistic"

    @staticmethod
    def solve_from_instances(group_rule: GroupRule, instances: List[object]):

        type_a = [inst for inst in instances if isinstance(inst, group_rule.types[0])]
        type_b = [inst for inst in instances if isinstance(inst, group_rule.types[1])]

        matrix = build_cost_matrix(group_rule, type_a, type_b)
        matching = bin_search(matrix)
        return recover_groups(matching, type_a, type_b)

    @staticmethod
    def solve_from_valid_groups(group_rule: GroupRule, groups: List[Group]):
        matrix = build_cost_matrix_from_groups(group_rule, groups)
        matching = bin_search(matrix)
        return recover_groups_from_groups(matching, groups)
```

Script C.1 – Hungarian Algorithm with Binary Search Solver Source Code

APPENDIX D – STABLE MARRIAGE SOLVER

```
from src.solvers.base_solver import Solver
from src.group import Group, GroupRule
from typing import List, Type


def check_preference(prefer, n, m0, m1):
    """
    Check if element N prefers element M0 over element M1.
    """
    return prefer[wn.index(m1)] < prefer[n].index(m0)


def make_stable_pairs(type_a, type_b):
    """
    Stable Marriage algorithm to find stable pairs between two types.
    """
    # Assume type_a and type_b have a 'preference' attribute that is a list of
        preferences
    for a in type_a:
        if not hasattr(a, 'preference'):
            raise ValueError(f"Instances of type {type(a).__name__} must have a '
                preference' attribute.")
    for b in type_b:
        if not hasattr(b, 'preference'):
            raise ValueError(f"Instances of type {type(b).__name__} must have a '
                preference' attribute.")

    # Build preference lists
    prefer = {a: a.preference for a in type_a}
    prefer.update({b: b.preference for b in type_b})

    w_partner = {b: None for b in type_b}  # Women's partners
    m_free = set(type_a)  # Free men

    while m_free:
        m = m_free.pop()
        for w in prefer[m]:
            if w_partner[w] is None:  # Woman is free
                w_partner[w] = m
                break
            else:  # Woman is already engaged
                m1 = w_partner[w]
                if not check_preference(prefer, w, m, m1):
                    w_partner[w] = m
                    m_free.add(m1)
                    break

    return [(m, w) for w, m in w_partner.items()]

def build_groups(stable_pairs):
    """
    Build groups from stable pairs.
    """
    groups = []
    for m, w in stable_pairs:
        group = Group()
        group.add_member(m)
```

```
            group.add_member(w)
            groups.append(group)
        return groups


class StableMarriage(Solver):
    """
    Solver using the Stable Marriage algorithm for assignment problems.
    """

    @staticmethod
    def can_solve(group_rule: GroupRule):
        """
        Check if the group rule can be solved using the Stable Marriage algorithm.
        """
        if len(group_rule.cardinality_rules) != 2:
            return False
        for _, (min_count, max_count) in group_rule.cardinality_rules.items():
            if min_count != 1 or max_count != 1:
                return False
        return group_rule.stable_match and group_rule.objective_function_name == "
            no_statistic"

    @staticmethod
    def solve_from_instances(group_rule: GroupRule, instances: List[object]):
        """
        Solve the problem using the Stable Marriage algorithm.
        """
        type_a = [inst for inst in instances if isinstance(inst, group_rule.types[0])]
        type_b = [inst for inst in instances if isinstance(inst, group_rule.types[1])]

        stable_pairs = make_stable_pairs(type_a, type_b)
        return build_groups(stable_pairs)

    @staticmethod
    def solve_from_valid_groups(group_rule: GroupRule, groups: List[Group]):

        unique_instances = {}
        for group in groups:
            for cls, instance in group.members.items():
                if cls not in unique_instances:
                    unique_instances[cls] = list()
                for i in instance:
                    if i not in unique_instances[cls]:
                        unique_instances[cls].append(i)

        [type_a, type_b] = unique_instances.values()

        stable_pairs = make_stable_pairs(type_a, type_b)
        return build_groups(stable_pairs)
```

Script D.1 – Stable Marriage Solver Source Code

APPENDIX E – GENETIC ALGORITHM SOLVER

```
from src.solvers.base_solver import Solver
from src.group import Group, GroupRule
from typing import List, Type
import numpy as np


def optimize_by_instances(instances: List[object], group_rule: GroupRule, n_groups=3,
    n_generations=10000, pop_size=40, mutation_prob=0.5) -> List[Group]:
    '''
    The chromosome is a binary vector of size n_groups * n_instances, representing a
        n_groups x n_instances matrix.
    An element can belong to more than one group.
    '''
    from deap import base, creator, tools, algorithms
    import random
    import numpy as np

    m = len(instances)
    n = n_groups

    creator.create("FitnessMax", base.Fitness, weights=(2.0,))
    creator.create("Individual", list, fitness=creator.FitnessMax)

    toolbox = base.Toolbox()

    def random_individual():
        # Binary vector of size n*m
        return [random.randint(0, 1) for _ in range(n * m)]

    toolbox.register("individual", tools.initIterate, creator.Individual,
        random_individual)
    toolbox.register("population", tools.initRepeat, list, toolbox.individual)

    def mate(parent1, parent2):
        # Uniform crossover per gene
        child1 = [parent1[i] if random.random() < 0.5 else parent2[i] for i in range(n *
            m)]
        child2 = [parent2[i] if random.random() < 0.5 else parent1[i] for i in range(n *
            m)]
        return creator.Individual(child1), creator.Individual(child2)

    def mutate(ind):
        for i in range(n * m):
            if random.random() < mutation_prob:
                ind[i] = 1 - ind[i]
        return (ind,)

    def eval_grouping(ind):
        # Converts vector to n x m matrix
        mat = np.array(ind).reshape((n, m))
        group_objs = []
        for i in range(n):
            members = [instances[j] for j in range(m) if mat[i, j] == 1]
            if members:
                g = Group()
                g.add_member(members)
                group_objs.append(g)
```

```
        score = 0
        # Validation
        for g in group_objs:
            if not group_rule.validate(g):
                score += -1e8
        # Statistics
        score += group_rule.objective_function(group_objs) if group_objs else 0
        return (score,)

    toolbox.register("evaluate", eval_grouping)
    toolbox.register("mate", mate)
    toolbox.register("mutate", mutate)
    toolbox.register("select", tools.selTournament, tournsize=3)

    pop = toolbox.population(n=pop_size)
    hof = tools.HallOfFame(1)

    algorithms.eaSimple(pop, toolbox, cxpb=0.7, mutpb=0.3, ngen=n_generations, halloffame
        =hof, verbose=False)

    best = hof[0]
    mat = np.array(best).reshape((n, m))
    group_objs = []
    for i in range(n):
        members = [instances[j] for j in range(m) if mat[i, j] == 1]
        if members:
            g = Group()
            g.add_member(members)
            group_objs.append(g)
    return group_objs

def optimize_by_groups(groups: List[Group], group_rule: GroupRule, n_generations=10000,
    pop_size=30, mutation_prob=0.2) -> List[Group]:
    """
    The chromosome is a bit mask b of size n, where n is the number of
    allowed Grouping Objects. The i-th bit in b indicates whether or not the i-th
        grouping object is
    selected at that solution option.
    """
    from deap import base, creator, tools, algorithms
    import random

    n = len(groups)

    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
    creator.create("Individual", list, fitness=creator.FitnessMax)

    toolbox = base.Toolbox()
    toolbox.register("attr_bool", random.randint, 0, 1)
    toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.
        attr_bool, n)
    toolbox.register("population", tools.initRepeat, list, toolbox.individual)

    def eval_selection(individual):
        selected_groups = [g for i, g in enumerate(groups) if individual[i]]
        score = 0
        # Statistics
        score = group_rule.objective_function(selected_groups) if selected_groups else 0
        return (score,)
```

```
    toolbox.register("evaluate", eval_selection)
    toolbox.register("mate", tools.cxTwoPoint)
    toolbox.register("mutate", tools.mutFlipBit, indpb=mutation_prob)
    toolbox.register("select", tools.selTournament, tournsize=3)

    pop = toolbox.population(n=pop_size)
    hof = tools.HallOfFame(1)

    algorithms.eaSimple(pop, toolbox, cxpb=0.7, mutpb=0.3, ngen=n_generations, halloffame
        =hof, verbose=False)

    best = hof[0]
    selected_groups = [g for i, g in enumerate(groups) if best[i]]
    return selected_groups

class GeneticAlgorithm(Solver):

    @staticmethod
    def can_solve(group_rule: GroupRule):
        return True

    @staticmethod
    def solve_from_instances(group_rule: GroupRule, instances: List[object]):
        answers = optimize_by_instances(instances, group_rule, n_groups=len(instances))
        return answers

    def solve_from_valid_groups(group_rule: GroupRule, groups: List[Group]):
        answers = optimize_by_groups(groups, group_rule)
        return answers
```

Script E.1 – Genetic Algorithm Solver Source Code

APPENDIX F – FAIR BIPARTITE MATCHING SOLVER

```
\begin{lstlisting}
from src.solvers.base_solver import Solver
from src.group import Group, GroupRule
from typing import List, Type

import igraph as ig
import matplotlib.pyplot as plt
import numpy as np
import networkx as nx
import random as rd

class quota:
    def __init__(self, characteristic, distribution, scale=True):
        self.characteristic = characteristic
        self.distribution = distribution
        total = sum([ x[1] for x in self.distribution ])
        if scale:
            map(lambda x: [x[0], x[1]/total] , self.distribution)
        elif total < 1:
            distribution.append(['Remaining', 1-total])

    def __str__(self):
        return "{ " + str(self.characteristic) + ": " + str(self.distribution) + " }"

    def __mul__(self, obj):
        if len(self.distribution) == 0:
            return obj

        if len(obj.distribution) == 0:
            return self

        new_distribution = []
        for distribution_1 in self.distribution:
            for distribution_2 in obj.distribution:
                new_distribution.append([distribution_1[0] + '+' + distribution_2[0],
                    distribution_1[1] * distribution_2[1]])
        return quota(self.characteristic + '/' + obj.characteristic, new_distribution)

class quotas_description:
    def __init__(self, quotas):
        self.requirement = self.combine_quotas(quotas)

    def __str__(self):
        return str(self.requirement)

    def characteristic(self):
        return self.requirement.characteristic

    def distribution(self):
        return self.requirement.distribution

    def combine_quotas(self, quotas):
        new_requirement = quota("", [])
        for requirement in quotas:
            new_requirement *= requirement
        return new_requirement
proxy={}
```

```
class mapper:
    def __init__(self, group_a, group_b, matches, quotas_group_a):
        self.group_a = group_a
        self.group_b = group_b
        self.matches = matches
        self.quotas_group_a = quotas_group_a
        self.number_of_matches = len(group_b)

        self.graph = self.build_graph()

    def build_graph(self):
        global proxy
        proxy={}
        g = ig.Graph(directed=True)
        G = nx.Graph()
        G.add_node(len(g.vs), subset=0, name="source", label='Source')
        g.add_vertex(type='source', name="source")


        remaining_matches = self.number_of_matches
        for group in self.quotas_group_a.requirement.distribution:
            G.add_node(len(g.vs), name=group[0], label='Fair Group', quotas=group[1],
                subset=1, obj=group)
            g.add_vertex(name=group[0], quotas=group[1], type='Fair Group', obj=group)
            G.add_edge(0, len(g.vs)-1, capacity=group[1], weight=0)
            g.add_edge(0, len(g.vs)-1, capacity=group[1], weight=0)
            remaining_matches-=group[1]

        # Remaining
        if len(self.quotas_group_a.requirement.distribution):
            G.add_node(len(g.vs), name="Remaining", label='Fair Group', subset=1, obj={'
                extra': True})
            g.add_vertex(name="Remaining", type='Fair Group', obj={'extra': True})
            G.add_edge(0, len(g.vs)-1, capacity=remaining_matches, weight=0)
            g.add_edge(0, len(g.vs)-1, capacity=remaining_matches, weight=0)

        for obj in self.group_a:
            G.add_node(len(g.vs), obj=obj, subset=3, name="Worker", label='Worker')
            g.add_vertex(obj=obj, type='group_a')

        self.add_edges_group_and_requirement(self.group_a, self.quotas_group_a, 0, g, G)


        for obj in self.group_b:
            G.add_node(len(g.vs), obj=obj, subset=4, name="Job", label='Job')
            g.add_vertex(obj=obj, type='group_b')


        for hash in self.matches:
            [[_, u], [_, v], [_, w]] = hash.items()
            G.add_edge(g.vs.find(obj=self.group_a[u]).index, g.vs.find(obj=self.group_b[v
                ]).index, capacity=1, weight=w)
            g.add_edge(g.vs.find(obj=self.group_a[u]).index, g.vs.find(obj=self.group_b[v
                ]).index, capacity=1, weight=w)

        G.add_node(len(g.vs), subset=6, name="target", label='Target')
        g.add_vertex(type='target', name="target")
```

```
        self.add_edges_group_and_requirement(self.group_b, quotas_description([]), len(g.
            vs)-1, g, G)


        return G

    def has_quotas(self, quotas):
        return len(quotas.requirement.distribution) >= 1

    def add_edges_group_and_requirement(self, group, requirement, in_case_its_empty, g, G
        ):
        list_of_characteristics = requirement.requirement.characteristic.split('/')
        if len(requirement.requirement.distribution) == 0:
            for obj in group:
                g.add_edge(in_case_its_empty, g.vs.find(obj=obj).index, capacity=1,
                    weight=0)
                G.add_edge(in_case_its_empty, g.vs.find(obj=obj).index, capacity=1,
                    weight=0)

        else:


            list_proxy=[]
            for dist in requirement.requirement.distribution:
                list_of_distribution = dist[0].split('/')
                for obj in group:
                    if not any([getattr(obj, list_of_characteristics[i], None) !=
                        list_of_distribution[i] for i in range(len(list_of_distribution))
                        ]):
                        if obj not in proxy.values():
                            key = rd.random()
                            G.add_node(len(g.vs), subset=2, obj=key, name="proxy", label=
                                'proxy')
                            g.add_vertex(name='proxy', type='proxy', obj=key)
                            g.add_edge(len(g.vs)-1, g.vs.find(obj=obj).index, capacity=1,
                                 weight=0, type='proxy')
                            G.add_edge(len(g.vs)-1, g.vs.find(obj=obj).index, capacity=1,
                                 weight=0, type='proxy')
                            proxy[key] = obj
                        g.add_edge(g.vs.find(obj=dist).index, len(g.vs)-1, capacity=1,
                            weight=0)
                        G.add_edge(g.vs.find(obj=dist).index, len(g.vs)-1, capacity=1,
                            weight=0)



            for obj in group:
                if obj not in proxy.values():
                    key=rd.random()
                    G.add_node(len(g.vs), subset=2, obj=key, name="proxy", label='proxy')
                    g.add_vertex(name='proxy', type='proxy', obj=key)
                    g.add_edge(len(g.vs)-1, g.vs.find(obj=obj).index, capacity=1, weight
                        =0, type='proxy')
                    G.add_edge(len(g.vs)-1, g.vs.find(obj=obj).index, capacity=1, weight
                        =0, type='proxy')
                    proxy[key] = obj
                proxy_obj = [i for i in proxy if proxy[i]==obj][0]
```

```
                g.add_edge(g.vs.find(obj={'extra': True}).index, g.vs.find(obj=proxy_obj)
                    .index, capacity=1, weight=0)
                G.add_edge(g.vs.find(obj={'extra': True}).index, g.vs.find(obj=proxy_obj)
                    .index, capacity=1, weight=0)

def update_graph(G, mincostFlow):
    for u in mincostFlow:
        for v in mincostFlow[u]:
            if u>v and G.edges[u, v].get('type') == 'proxy' and  G.edges[u, v]["used"
                ]==0:
                 G.edges[u, v]["used"] = mincostFlow[u][v]
            if u > v: continue
            G.edges[u, v]["used"] = mincostFlow[u][v]

def solve(G):
    mincostFlow = nx.max_flow_min_cost(G, 0, len(G.nodes)-1)
    update_graph(G, mincostFlow)
    return[nx.maximum_flow_value(G, 0, len(G.nodes)-1), nx.cost_of_flow(G, mincostFlow)]


def gen_matching_from_instances(gr, type_a, type_b):
    list = []
    for source in range(len(type_a)):
        for destiny in range(len(type_b)):
            group = Group()
            group.add_member(type_a[source])
            group.add_member(type_b[destiny])
            list.append({
                "source": source,
                "destiny": destiny,
                'weight': gr.statistics[0](group.members)
            })
    return list

def gen_matching_from_groups(gr, groups):
    list = []
    for group in groups:
        for a in group.members[0]:
            for b in group.members[1]:
                list.append({
                    "source": a.ID,
                    "destiny": b.ID,
                    'weight': gr.statistics[0](group.members)
                })
    return list

def build_groups(m):
    used_edges = [(u, v) for u, v, d in m.graph.edges(data=True) if 'used' in d and d['
        used'] > 0]
    global proxy
    fair_pairs = []
    for u, v in used_edges:
        if m.graph.nodes[u]['label'] == 'Worker' and m.graph.nodes[v]['label'] == 'Job':
            fair_pairs.append((m.graph.nodes[u]['obj'], m.graph.nodes[v]['obj']))

    groups = []
    for a, b in fair_pairs:
        group = Group()
        group.add_member(a)
```

```
        group.add_member(b)
        groups.append(group)
    return groups

class FairBipartiteMatching(Solver):
    """
    Solver using the Fair Bipartite algorithm for assignment problems.
    """

    @staticmethod
    def can_solve(group_rule: GroupRule):
        if len(group_rule.cardinality_rules) != 2:
            return False
        for _, (min_count, max_count) in group_rule.cardinality_rules.items():
            if min_count != 1 or max_count != 1:
                return False

        # check if there is a validator called "fair_matching"
        return len(group_rule.quotas) > 0

    @staticmethod
    def solve_from_instances(group_rule: GroupRule, instances: List[object]):

        quotas = quotas_description([quota(quota_name, quota_definition) for quota_name,
            quota_definition in group_rule.quotas.items()])

        type_a = [inst for inst in instances if isinstance(inst, group_rule.types[0])]
        type_b = [inst for inst in instances if isinstance(inst, group_rule.types[1])]

        matching = gen_matching_from_instances(group_rule, type_a, type_b)
        m = mapper(type_a, type_b, matching, quotas)
        solve(m.graph)

        return build_groups(m)


    @staticmethod
    def solve_from_valid_groups(group_rule: GroupRule, groups: List[Group]):
        unique_instances = {}
        for group in groups:
            for cls, instance in group.members.items():
                if cls not in unique_instances:
                    unique_instances[cls] = list()
                for i in instance:
                    if i not in unique_instances[cls]:
                        unique_instances[cls].append(i)

        [type_a, type_b] = unique_instances.values()

        matching = gen_matching_from_groups(group_rule, groups)

        m = mapper(type_a, type_b, matching, quotas)
        solve(m.graph)

        return build_groups(m)
```

Script F.1 – Fair Bipartite Matching Solver Source Code

UDESC UNIVERSIDADE DO ESTADO DE SANTA CATARINA | JOINVILLE CENTRO DE CIÊNCIAS TECNOLÓGICAS

UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
BIBLIOTECA UNIVERSITÁRIA
REPOSITÓRIO INSTITUCIONAL

CENTRO DE CIÊNCIAS TECNOLÓGICAS – CCT

ATESTADO DE VERSÃO FINAL

Eu, Yuri Kaszubowski Lopes, professor(a) do curso de Mestrado Acadêmico em Computação Aplicada, declaro que esta é a versão final aprovada pela comissão julgadora da dissertação/tese intitulada: **"Categorised Grouping: A framework for industry applications"** de autoria do(a) acadêmico Rafael Granza de Mello.

Joinville, 4 de setembro de 2025.

Assinatura digital do(a) orientador(a):

gov.br Documento assinado digitalmente
YURI KASZUBOWSKI LOPES
Data: 03/09/2025 18:29:14-0300
Verifique em https://validar.iti.gov.br

Yuri Kaszubowski Lopes