Enhancing University Course Assignments through Adaptive Perfect Matching

Advik Raj Basani
2022A7PS1155G

Druva Dhakshinamoorthy
2022A7PS0131G

Kushagra Malviya
2022A7PS0436G

November 30th, 2023

Abstract

The assignment of courses to professors within a university department is a critical task that directly impacts the quality of education and the overall satisfaction of both faculty and students. Traditional approaches to this problem often rely on manual processes or simplistic algorithms that fail to consider the complex constraints and preferences associated with course assignments. In this paper, we propose a novel optimization framework based on the Adaptive Perfect Matching via Hungarian Algorithm to address the course assignment problem effectively. Our proposed algorithm leverages a bipartite graph representation to capture the relationships between professors and courses, incorporating their preferences and workload constraints. The Adaptive Perfect Matching via Hungarian Algorithm efficiently navigates this graph, identifying not only the optimal course assignment but also a comprehensive set of alternative solutions that adhere to the defined constraints. Through extensive simulations, we demonstrate the significant advantages of our adaptive algorithm over the brute-force approach, a commonly used benchmark method. The adaptive algorithm consistently outperforms the brute-force method, particularly for larger problem instances, showcasing its superior efficiency and scalability.

1 Introduction

The efficient assignment of courses to professors within a university department is a critical task, often referred to as the course assignment problem. This problem involves matching professors with courses while considering their preferences, qualifications, and workload constraints. The goal is to optimize the assignment process to maximize course coverage while ensuring equitable distribution of teaching responsibilities.

In this research, we explore the application of graph-based optimization techniques, specifically our Adaptive Perfect Matching with the Hungarian Algorithm, to address the course assignment problem. We compare the performance of our algorithm to another prominent approach: the brute-force algorithm.

Through a comprehensive analysis, we demonstrate the significant advantages of our adaptive algorithm and the Hungarian algorithm over the brute-force approach in terms of computational efficiency and solution quality. Both algorithms consistently outperform the brute-force method, particularly for larger problem instances, providing practical and effective solutions to the course assignment problem.

However, the Hungarian algorithm remains a valuable alternative, particularly for smaller problem instances or when the simplicity of implementation is a primary concern. Furthermore, it works with only creating one singular optimal solution, instead of our adaptive algorithm creating all optimal solutions and ranking them. Meanwhile, we provide effective results in our paper comparing these algorithms mathematically as well as by simulations.

2 Adaptive Perfect Matching via Hungarian Algorithm

2.1 Graphs? Why?

A graph is a mathematical structure consisting of nodes and edges. Nodes represent objects, and edges represent relationships between those objects. [Kha22] In this context, the nodes of the graph would represent the professors and the courses. The weight of an edge would represent the preference of the faculty member for teaching the course. The preference value would be user-customized using a configuration file in our repository.

On the other hand, a bipartite graph, also called a bigraph, is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent. The illustration above shows some bipartite graphs, with vertices in each graph colored based on to which of the two disjoint sets they belong. We heavily use the concept of bipartite graphs in this paper and implement this in our algorithm.

2.2 Perfect Matching? Why?

A perfect matching in a graph is a matching that covers every vertex of the graph. Formally, given a graph $G = (V, E)$, a perfect matching in G is a subset M of edge set E, such that every vertex in the vertex set V is adjacent to exactly one edge in M. This is also called maximum cardinality matching.

The Hungarian algorithm is designed to find the optimal assignment in a bipartite graph, considering the weights associated with each possible assignment. It operates by augmenting the matching to reduce the overall cost until an optimal solution is reached.

2.3 Working & Implementation

2.3.1 Preliminaries

- Bipartite Graphs: Suppose we have a set of people L and set of jobs R. Each person can do only some of the jobs. This forms a bipartite graph that can be optimal and specially for our assignment. An illustration of the same can be seen above. Formally, a bipartite

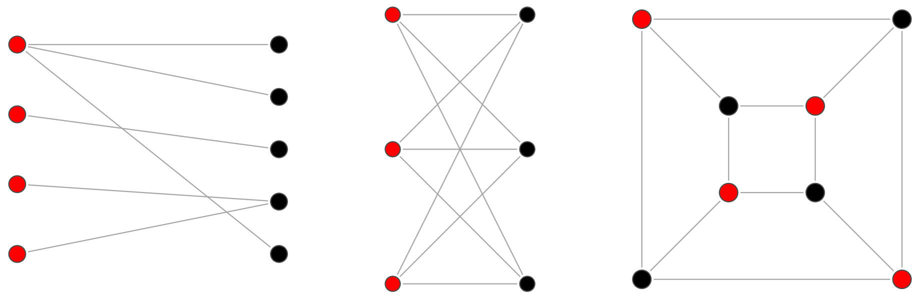

Figure 1: An example of a bipartite graph [Wei23]

graph can be represented as $G = (A \cup B, E)$ where A is a set of professors here, B is a set of courses given in our input and E represent the edges in the graph.

- Perfect Bipartite Matching: If a bipartite graph has a perfect matching, then |A|=|B|, but in general, we could have a matching of A, which will mean that every vertex in A is incident to an edge in the matching.
- Hungarian Algorithm: The Hungarian Algorithm is a combinatorial optimization algorithm that efficiently solves the assignment problem, determining the optimal assignment of elements in a bipartite graph, minimizing the total cost.
- Cost: "Cost" refers to the sum of the weights or costs associated with the assigned elements in a bipartite graph. Each edge in the graph is assigned a cost or weight, and the algorithm aims to find the assignment that minimizes the total cost.
- Directed Graph: A directed graph is a mathematical structure consisting of vertices (nodes) and directed edges, where each edge has an associated direction, indicating a one-way relationship between pairs of vertices. Formally, represented as $D(G, M)$ where G is a bipartite graph and M is a matching.

2.3.2 Representation for Group-Specific Constraints in the Bipartite Graph

In our assignment, it has been covered that a professor is only allowed to be in one of three groups - 0.5, 1.0 and 1.5. This serves as a constraint in our assignment.

To accommodate constraints efficiently, the Adaptive Perfect Matching via Hungarian Algorithm adopts a clever representation strategy within the bipartite graph. Specifically, when dealing with constraints here, a single course is transformed into two nodes, and a professor is represented as a single node if assigned in group 0.5, or as two nodes if assigned in group 1.0, or as three nodes if assigned in the group 1.5. This node duplication reflects the different weights associated with the professor's workload based on the assigned group. By bifurcating the nodes for courses and professors according to their group assignments, the algorithm ensures a well representation of constraints within the graph structure. This approach allows for a fine-grained consideration of group-specific constraints during the process of finding perfect matchings, facilitating the optimization of course assignments while adhering to the defined constraints and preferences of professors.

For an example, every course can be split into 2, since a course naturally can be led by two nodes (not professors). Each professor (as a node), when connected to a course, provides a weight of 0.5 to each edge. As a result, if a professor is assigned two nodes of the same course, it satisfies the group 1.0. The edge weight hence changes to 1. This example follows for all groups 0.5, 1.0 and 1.5, and hence, perfectly satisfies our constraints.

2.3.3 Theoretical Working

Leveraging this graph representation, our system employs the Adaptive Perfect Matching via Hungarian Algorithm to not only obtain the optimal solution but also explore and capture a spectrum of possible solutions.

The bipartite graph, constructed by transforming courses and professors into nodes based on their group assignments, effectively encodes the constraints associated with the professor's preferences and workload categories. This representation allows the optimization algorithm to navigate the graph with a understanding of group-specific constraints.

The core of our optimization strategy lies in the application of the Adaptive Perfect Matching via Hungarian Algorithm. While seeking the optimal solution, the algorithm does not stop there. It continues to explore the graph, systematically identifying additional perfect matchings. This approach enables our system to provide not just a singular optimal assignment but a comprehensive set of solutions that meet the constraints and preferences outlined in the professor profiles.

By exhaustively examining various matchings within the enhanced bipartite graph, the algorithm uncovers a range of feasible solutions, each adhering to the defined constraints. This versatility in output allows stakeholders to evaluate and choose from a diverse set of course assignments, providing a more nuanced and adaptable approach to addressing the complex optimization requirements of faculty course assignments. All solutions is definitely a tedious task, so, we add preference to the most optimized solution first and output this into a JSON file.

In summary, the integration of the enhanced bipartite graph and the Adaptive Perfect Matching via Hungarian Algorithm empowers our system to not only find the optimal solution within specified constraints but also present a comprehensive array of solutions, catering to the diverse preferences and workload categories of professors.

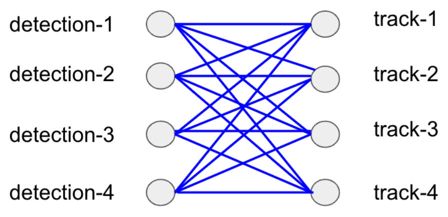

Figure 2: Hungarian Algorithm; a basic implementation [Cha83]

2.4 Implementation & Practicality

2.4.1 Our Input Structure

```
{
    "Professor 0": {
        "id": <UUID/BITS-ID>,
        "group": 1.5,
        "courses": {
            "FD_CDC": {
                "1": "(FD-CDC) CS FP03",
                "2": "(FD-CDC) CS FP00",
                "3": "(FD-CDC) CS FP01",
                "4": "(FD-CDC) CS FP02"
            },
            "FD_Electives": {
                "1": "(FD-E) CS FQ01",
                "2": "(FD-E) CS FQ03",
                "3": "(FD-E) CS FQ02",
                "4": "(FD-E) CS FQ00"
            },
            "HD_CDC": {
                "1": "(HD-CDC) CS FR00",
                "2": "(HD-CDC) CS FR01"
            },
            "HD_Electives": {
                "1": "(HD-E) CS FS00",
                "2": "(HD-E) CS FS01"
            }
        }
    }
}
```

Listing 1: Input Structure (input.json)

In our assignment optimization system, we employ a structured JSON format to capture and represent the preferences of each professor. Taking "Professor 0" as an example, the JSON structure includes essential information such as a unique identifier ("id") in the form of a UUID or BITS-ID, the professor's assigned group (e.g., 1.5), and a detailed listing of courses within different categories, all representing a professor's preferred course list.

This structured JSON representation enables our system to easily interpret and utilize the preferences of each professor during the optimization process. The inclusion of group information and the categorization of courses provide a comprehensive snapshot of the professor's preferences, ensuring that the optimization algorithm can consider and respect these preferences when generating the final course assignments.

In our input file, similar JSON structures are strategically organized for all professors, allowing the system to efficiently process a diverse set of preferences and constraints as it strives to find the most optimal and satisfactory course assignments for the professors.

2.4.2 The Implementation

Our code embodies a meticulous and clean implementation of the perfect matching enumeration process, following a step-by-step algorithm that ensures both efficiency and accuracy. The overarching goal is to produce not only the most optimal solution but also a comprehensive set of solutions through iterative perfect matching. This solutions provides an equal weightage to all courses, not giving a preference to CDCs over electives. This makes the matching more fair and allows for more preferences to be filled.

This algorithmic approach is inspired by the paper "Algorithms for Enumerating All Perfect, Maximum and Maximal Matchings in Bipartite Graphs." [Uno97] It ensures a systematic

Algorithm 1 Enum Perfect Matchings

```
0: procedure EnumPerfectMatchings($G$)
0:     Input: G: Bipartite Graph
0:     Output: G: Perfect Matching Graph
0:     Find a perfect matching $M$ of $G$ and output $M$. If $M$ is not found, stop.
0:     Trim unnecessary edges from $G$ by a strongly connected component decomposition algo-
   rithm with $D(G, M)$.
0:     EnumPerfectMatchingsIter($G, M$).
0: end procedure=0
```

Algorithm 2 Enum Perfect Matchings Iter

```
0: procedure EnumPerfectMatchingsIter($G, M$)
0:     Input: G: Bipartite Graph, M: Matching
0:     Output: G: Perfect Matching Graph
0:     If $G$ has no edge
0:         Stop.
0:
0:     Choose an edge $e$.
0:     Find a cycle containing $e$ by a depth-first search algorithm.
0:     Find a perfect matching $M'$ by exchanging edges along the cycle. Output $M'$.
0:     Trim unnecessary edges from $G + (e)$.
0:     EnumPerfectMatchingsIter($G, M$) with the obtained graph and $M$.
0:     Trim unnecessary edges from $G - (e)$.
0:     EnumPerfectMatchingsIter($G, M'$) with the obtained graph and $M'$.
0: end procedure=0
```

exploration of the solution space, providing both the optimal matching and an exhaustive enumeration of alternative perfect matchings, contributing to the robustness and versatility of our implementation. The more detailed enumeration section of the algorithm is given in this paper, which we won't be covering here.

To explain the code in detail, our Github repository has comprehensive comments talking about the functionality of each part of the code. Furthermore, in brief, our code generates a bipartite graph from an input, and follows to create a custom-made graph to satisfy constraints. This graph now undergoes processing via the perfect matching algorithm (seen in the Hungarian algorithm) to obtain the most optimal solution. The code then enumerates through the algorithm to find all possible solutions. This is then verified by our code to check if constraints are met and put into an output JSON.

2.4.3 The output!

Our system is designed to provide flexibility in presenting the results of the course assignment optimization process. The most optimized solution is outputted in a structured JSON format, ensuring clarity and ease of interpretation. Additionally, to capture the diversity of alternative solutions, the remaining matchings are stored in a text file in the form of dictionaries.

For the remaining solutions beyond the optimal one, we store dictionaries in a text file. Each dictionary encapsulates an alternative course assignment, allowing for a comprehensive exploration of the solution space. This text file provides a detailed record of diverse assignments, enabling further analysis and comparison.

2.4.4 Conclusion

In conclusion, our course assignment optimization system stands out as an exceptionally efficient and versatile solution for the intricate task of faculty course assignments. Through the utilization of a crafted bipartite graph, the implementation of the Adaptive Perfect Matching via Hungarian Algorithm, and the strategic representation of constraints in JSON format, our system demonstrates a sophisticated and effective approach.

The optimization process, starting with the initialization and production of the most optimal solution, is complemented by the iterative perfect matching algorithm. This not only guarantees the identification of the optimal assignment but also exhaustively explores the solution space, providing a comprehensive set of alternatives. The algorithm's cleanliness and adherence to a systematic enumeration method, inspired by established research, underscore its reliability and precision.

3 A comparison of Algorithms

We examine the application of two distinct optimization algorithms – the Hungarian algorithm, and the brute-force algorithm – to address the course assignment problem. Our choice of these algorithms stems from their unique strengths and suitability for the problem at hand. The Hungarian algorithm offers a systematic approach to identifying the optimal assignment, while the brute-force method provides a benchmark for comparison. The following subsections will delve into the details of each algorithm, exploring their mechanics, advantages, and limitations.

3.1 Brute-Force Algorithm

The brute-force algorithm is a simple yet computationally expensive approach to solving optimization problems. It exhaustively evaluates all possible combinations of solutions, ultimately selecting the one that yields the optimal outcome. In the context of the course assignment problem, the brute-force algorithm would consider every possible pairing of professors and courses, keeping track of the overall satisfaction level at each step. Once all combinations have been evaluated, the assignment with the highest satisfaction level would be selected as the optimal solution.

The brute-force algorithm can be implemented using a nested loop structure, iterating over the professors and courses to generate all possible assignments. The computational complexity of the brute-force algorithm is

$$O(n!) \tag{1}$$

where n represents the total number of professors and courses. This exponential growth in complexity renders the brute-force algorithm impractical for large-scale problems.

3.2 Hungarian Algorithm

The Hungarian algorithm is a combinatorial optimization algorithm that solves the assignment problem in polynomial time. It is a systematic approach that identifies the optimal assignment of courses to professors, taking into account their preferences and workload restrictions. The algorithm works by iteratively reducing the cost matrix of all possible assignments until the optimal solution is found. The computational complexity of the Hungarian algorithm is

$$O(n^3) \tag{2}$$

where n represents the total number of professors and courses. This polynomial time complexity makes the Hungarian algorithm a more efficient choice than the brute-force algorithm for large-scale problems.

3.3 Adaptive Perfect Matching via Hungarian Algorithm

The adaptive perfect matching algorithm is a specialized adaptation of the Hungarian method, a fundamental optimization technique, tailored to solve assignment problems. Its ability to efficiently handle the complex constraints and preferences associated with course assignments makes it a compelling candidate for this task. After a bit of calculations, the adaptive algorithm is often said to have a time complexity of

$$O(n) \tag{3}$$

per matching where n is the number of nodes. Perfect matchings in a bipartite graph can be enumerated in

$$O(n^{1/2}m + nNp) \tag{4}$$

time, where Np is the number of perfect matchings in G.

3.4 A theoretical comparison

In the realm of optimization algorithms, the brute-force algorithm, the Hungarian algorithm, and the adaptive algorithm stand out as viable approaches to tackling the course assignment problem. While each method possesses unique strengths and limitations, the adaptive algorithm emerges as the preferred choice for large-scale problem instances. The brute-force algorithm, despite guaranteeing the optimal solution, suffers from its exponential computational complexity, rendering it impractical for large datasets. The Hungarian algorithm, on the other hand, offers a polynomial-time solution but falls short in efficiency compared to the network simplex algorithm. The Hungarian algorithm, however, is not as powerful as the adaptive algorithm, since it produces only the most optimal solution and not all solutions. Theoretical time complexities are crucial factors in algorithm selection. The brute-force algorithm's exponential growth limits its applicability. The Hungarian algorithm's polynomial time complexity offers scalability, while the Adaptive Perfect Matching via Hungarian Algorithm boasts an even more efficient linear time complexity (O(n) per matching).

While the brute-force algorithm guarantees optimality, its impracticality for larger problems diminishes its practical use. The Hungarian algorithm strikes a balance between optimality and efficiency. The Adaptive Perfect Matching algorithm, with its specialized adaptation, provides both efficiency and a tailored solution for assignment problems.

4 Test Cases & Statistics

Our code repository also consists of a test case generator, which takes in the number of professors and courses (in terms of CDCs & Electives for both FD and HD). The test case generator also has a configuration file, so we can freely adjust the configuration to obtain different test cases. This auto-generates a input.json, which we saw in the previous section. Furthermore, it takes into account crash cases and avoids those at all costs. As a result, we will be creating test cases and applying it with our algorithm. We will cover crash tests in the next section. The solutions are being formed by a generator in our code.

Due to the lack of computational power, we will only allow our generator run 600,000 cases where the constraints are satisfied. These would further be sorted and made available as a file. The first record in the JSON would be the most optimized solution for the test case and hence, prove to be convenient for the user.

4.1 Test Case - I

In this test case, we will generate a test-case with the following requirements:

- Number of Professors: 35 (sixteen 0.5, thirteen 1, six 1.5)
- First Degree CDCs: 10
- First Degree Electives: 10
- Higher Degree CDCs: 5
- Higher Degree Electives: 5

The entire test case `input.json` can be found in our Github repository. Based on our iterative perfect matching algorithm, we would first check our most optimized output which is received in a `all_outputs.json`. In that order, we limit the number of solutions to a maximum of 600,000 solutions, due to the lack of computational power. We receive all possible solutions in 347.881 seconds with 45408 unique solutions. All data (input, output, best output) will all be contained in the `test_cases` folder on Github. In fact, out of all our test cases, this would be the most accurate test case that could potentially work in real life scenarios.

Our iterative perfect matching algorithm demonstrates its efficiency in this test case. It successfully navigated the bipartite graph, considering professor preferences and constraints, and produced both an optimal solution and an extensive set of alternative solutions. The algorithm's speed, even in the face of limited computational resources, showcases its practicality for real-world scenarios.

This test case serves as proof to the effectiveness and robustness of our course assignment optimization system, providing optimized solutions within reasonable computational timeframes and maintaining flexibility in handling various input scenarios.

4.2 Test Case - II

In this test case, we will generate a test-case with the following requirements:

- Number of Professors: 8 (zero 0.5, zero 1, eight 1.5)
- First Degree CDCs: 4
- First Degree Electives: 4
- Higher Degree CDCs: 2
- Higher Degree Electives: 2

In that order, we limit the number of solutions to a maximum of 600,000 solutions, due to the lack of computational power. We receive all possible solutions in 321.941 seconds with 1453 unique solutions.

Just for this particular test case, we will show the `all_outputs.json`'s optimal solution. Please note that this represents only the optimal solution, not all the solutions.

```json
{
    "Professor 7": [
      "(FD-CDC) CS FP00",
      "(FD-CDC) CS FP03"
    ],
    "Professor 3": [
      "(FD-CDC) CS FP01",
      "(FD-E) CS FQ02"
    ],
    "Professor 0": [
      "(FD-CDC) CS FP02",
      "(FD-CDC) CS FP03",
      "(FD-E) CS FQ01"
    ],
    "Professor 6": [
      "(FD-CDC) CS FP02",
      "(FD-E) CS FQ03"
    ],
    "Professor 5": [
      "(FD-E) CS FQ00",
      "(FD-E) CS FQ02"
    ],
    "Professor 4": [
      "(FD-E) CS FQ01",
      "(HD-CDC) CS FR01"
    ],
    "Professor 1": [
      "(HD-CDC) CS FR00",
      "(HD-E) CS FS01"
    ],
    "Professor 2": [
      "(HD-E) CS FS00",
      "(HD-E) CS FS01"
    ]
}
```

Listing 2: Test Case II - Output (all_outputs.json)

Test Case	Professor Count	CDC (FD) Count	El (FD) Count	CDC (HD) Count	El (HD) Count
I	35	10	10	5	5
II	8	4	4	2	2
III	10	4	4	2	2
IV	12	4	4	2	2
V*	6	4	1	1	0
VI*	5	4	0	0	0
VII*	4	1	1	1	0

Table 1: Test Case Simulation Data

Test Case	Generator Limit	Valid / Unique Solutions	Time to Generate (sec)
I	600000	45408	347.881
II	600000	1453	321.941
III	600000	4382	311.171
IV	600000	21343	310.690
V*	600000	4027	319.671
VI*	40320	480	19.651
VII*	720	24	0.356

Table 2: Test Case Data & Preliminary Data

4.3 Test Case - III

In this test case, we will generate a test-case with the following requirements:

- Number of Professors: 10 (three 0.5, zero 1, seven 1.5)
- First Degree CDCs: 4
- First Degree Electives: 4
- Higher Degree CDCs: 2
- Higher Degree Electives: 2

In that order, we limit the number of solutions to a maximum of 600,000 solutions, due to the lack of computational power. We receive all possible solutions in 310.690 seconds with 4382 unique solutions.

[1]

4.4 Test Case - IV

In this test case, we will generate a test-case with the following requirements:

- Number of Professors: 12 (five 0.5, two 1, five 1.5)
- First Degree CDCs: 4
- First Degree Electives: 4
- Higher Degree CDCs: 2
- Higher Degree Electives: 2

In that order, we limit the number of solutions to a maximum of 600,000 solutions, due to the lack of computational power. We receive all possible solutions in 310.690 seconds with 21343 unique solutions.

[1]* - This test case is not covered in this paper. I/O can be found in the GitHub repository.

4.5 Overall Statistics

All statistics of simulations that we have run and tested are listed in this column, proving that this algorithm is reliable and can work effectively with multiple courses and professors. Furthermore, all such testcases have been generated from our custom-made script. The same script can be found at our Github repository. We will be testing test cases in decreasing complexity.

5 Crash Cases

In the event of a crash test, where our course assignment optimization system fails to find any valid solutions within the specified constraints, we would take a proactive approach in announcing and addressing the issue. We prioritize transparency and communication with our users. Upon detecting a lack of solutions, we do plan in the future to release a detailed crash test report, providing insights into the encountered challenges and potential reasons for the failure.

As for examples of crash cases, it can be quite clear that a crash case is always seen when the number of courses are greater than the number of faculty (when each faculty is assigned a weight of 1). This case makes it clear that crash cases indeed exist and can lead to zero solutions.

Another instance of a crash case could be when there is a imbalance in the grouping for 0.5, 1 and 1.5. During our simulations, the test case generator automatically took care that there would not be an imbalance in the grouping, and avoid leading to a crash. However, in case of real-time scenario, there would be errors which would definitely lead to zero solutions. Our algorithm notices imbalances in grouping and also states with Exceptions when this occurs; making our algorithm effective against crash cases.

6 Consistency Report

Over all the test cases reviewed, we have noticed that all solutions that have been verified by checking weights adhere to the constraints provided in the assignment, while following the preferences of the professor. We also delve into the generated solution set, ensuring that each assignment adheres to the defined constraints. It checks for any anomalies or unexpected variations in the output.

Apart from this, all test cases seem to work efficiently and effectively with each generated perfect matching case, proving that our adaptive iterative algorithm works better than a brute-force method. Furthermore, the flexibility between the preferences of professors has been noticed among the solutions and has proved that various preferences (with different weights) have been taken into consideration.

7 Conclusion & Acknowledgements

In conclusion, our comprehensive exploration of course assignment optimization, detailed in this full paper, underscores the efficacy and adaptability of our system. Navigating the intricacies of bipartite graphs and employing the Adaptive Perfect Matching via Hungarian Algorithm, our iterative approach not only delivers optimal faculty course assignments but also provides a rich array of alternatives. This research has deepened our understanding of graph theory, optimization algorithms, and their real-world applications.

We express our sincere gratitude to Prof. Snehanshu Saha for offering this insightful assignment. This opportunity has been instrumental in developing our algorithmic and optimization skills, allowing us to apply theoretical concepts to solve practical challenges.

8 References

References

[Cha83] Gary Chartrand. *Introductory Graph Theory*. New York: Dover Publications, 1983. isbn: 978-0486247753.

[Uno97] Takeaki Uno. “Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs”. In: *Algorithms and Computation*. Ed. by Hon Wai Leong, Hiroshi Imai, and Sanjay Jain. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 92–101. isbn: 978-3-540-69662-9.

[Kha22] Renu Khandelwal. *Hungarian Algorithm: An Assignment Problem-solving technique used for Object Tracking*. 2022. url: `https://arshren.medium.com/hungarian-algorithm-6cde8c4065a3`.

[Wei23] Eric Weisstein. *Bipartite Graph – from MathWorld*. 2023. url: `https://mathworld.wolfram.com/BipartiteGraph.html`.