Tutor Assignment and Coverage Optimization

Marena Keys, Macy Liang-Jones, Julia Yoon

December 5, 2025

Abstract

The Student Academic Success Center (SASC) at Carnegie Mellon University provides academic support for students through programs such as peer tutoring . Our project aims to improve the peer tutoring assignment process by developing a two-stage optimization model with two goals: better matching course demand and improving workload balance between tutors. Stage 1 maximizes assignment quality by aligning tutors with their preferred courses, while Stage 2 introduces equity considerations to distribute workloads. Using data from the tutor preference survey and course demand estimates, the model effectively meets the first goal by assigning tutors to courses aligned with their expertise, although workload balance is only partially achieved. These results suggest that the model captures realistic tutoring patterns while also highlighting the need for additional data and stronger equity constraints. Overall, this framework provides a foundation for more efficient tutor assignments for SASC.

Table of Contents

1 Introduction/Background **1**

2 Assumptions **2**

3 Problem Formulation **3**

3.1 Shared Data and Variables 3

3.2 Stage 1: Assignment Value Optimization 3

3.3 Stage 2: Workload Balance Optimization 4

4 Data Collection **6**

5 Implementation **7**

5.1 Handling of Infeasible Assignments 7

5.2 Data Pre-processing and Feasibility Filtering 8

5.3 Model Execution 8

6 Results **10**

6.1 Model Optimization 10

6.2 Tutor Assignments 11

6.3 Optimality Plots 13

6.4 Interpretation of Final Results 14

7 Limitations and Future Work **15**

8 Acknowledgements **16**

A Appendix: Code **17**

1 Introduction/Background

The Student Academic Success Center (SASC) at Carnegie Mellon University provides academic support and resources designed to help students develop the skills and strategies necessary for success in their coursework. SASC offers a variety of services, including academic coaching, tutoring, supplemental instruction, and workshops focused on effective study habits, time management, and collaborative learning. The center emphasizes a holistic approach to academic growth, encouraging students to build on their strengths while addressing challenges in a supportive, inclusive environment. They offer both individual sessions and group programs to accommodate all learning preferences that students may prefer.

However, it has been observed that the current process for assigning tutors to specific courses could be completed more efficiently. Although the center aims to align the tutors' expertise with courses offered, the current matching process lacks consistency and transparency. Some tutors are assigned to courses that may not fully utilize their strengths or prior experience, and others with similar qualifications may receive significantly different workloads. Some tutors work significantly longer hours than other students, which does not effectively distribute the work despite similar qualifications.

These discrepancies not only lead to an uneven distribution of tutoring hours but can also affect the overall quality and accessibility of academic support offered to students. For example, certain high-demand courses may experience a shortage of qualified tutors, while others have more coverage than necessary. This would be a prime example that leads to certain tutors working more hours in comparison to other tutors, and therefore a larger gap in workload between tutors that may also have similar qualifications but assigned different courses. This imbalance highlights the need for a more structured system for tutor-course assignments.

2 Assumptions

In order to focus on a specific assignment problem to model and optimize, we will be working on optimizing individual tutoring assignments only. We decided to optimize individual tutoring assignments because they are more geared toward specific students or goals in comparison to general help. This would therefore require the tutor assigned to the course in which the student is seeking help to be the best fit for this course or related topics. For instance, if a student is seeking help in a mathematics course, it would be a better fit for a math major student to be the assigned tutor in comparison to a student with a different major, even if they received the same grade in the class. The math major would have skills, advice, or any other information, and therefore be more equipped to help the student.

One consideration for this assignment is that the number of tutors hired cannot change. This implies that we are not maximizing which tutors to hire initially from a pool of students that are equipped to be tutors, but solving for a method to effectively assign a set of tutors considering these different parameters. Each tutor should be assigned a minimum amount of work, and no tutor should have a significantly higher amount of work than others. In addition, this notation of the "best fit" for a course will be information that the tutors themselves will provide, specifically what their top choice courses are followed by their secondary choice courses. This will be shown in later sections with more detail to reflect the bias towards the courses that tutors prefer to be assigned to.

3 Problem Formulation

To optimize tutor assignment and coverage, we used a two stage process that first optimizes for assignment quality and then optimizes for workload balance among tutors. We chose this order because assignment quality is the highest priority.

The model assigns tutors from set I to a set of courses J. A critical constraint of the real-world problem is that assignments are only possible if a tutor is qualified to teach a specific course. To ensure tutors are only assigned to courses they are qualified for, this constraint is implicitly managed through the Hour Cost matrix (H). For any tutor i who is not qualified to teach course j, the corresponding time cost H_{ij} is set to infinity. This formulation makes an unqualified assignment $x_{ij} = 1$ impossible, as it would instantly violate the tutor's maximum 10-hour workload constraint.

3.1 Shared Data and Variables

- I: Set of tutors
- J: Set of courses
- v_{ij}: Value score for assigning tutor i to course j
- d_j: Required demand (workload) for course j
- H_{ij}: The hour cost (hourly workload) per tutor for course j. If tutor i cannot tutor course j then $\text{time}_{ij} = \infty$.
- x_{ij} : 1 if tutors i is assigned to course j, 0 otherwise

3.2 Stage 1: Assignment Value Optimization

This first stage ensures that tutoring assignments are the highest quality possible by maximizing the total value score across all tutor-course assignments. It focuses only on finding the highest total assignment value (ie. value of placing a Math major with a Math course). This step establishes the best value standard, which will be used as a requirement in the next stage that establish workload equity between tutors.

Objective: Maximize $\sum_{i \in I} \sum_{j \in J} v_{ij} x_{ij}$

Subject to:

- **Tutor Workload Capacity:** Each tutor works maximum 10 hours a week

$$\sum_{j \in J} H_{ij} x_{ij} \leq 10 \qquad \forall i \in I$$

- **Course Covering:** Each course j gets its required demand

$$\sum_{i \in I} H_{ij} x_{ij} \geq d_j \qquad \forall j \in J$$

- **Tutor Course Minimum:** Each tutor is assigned at least 2 courses

$$\sum_{j \in J} x_{ij} \geq 2 \qquad \forall i \in I$$

- $x_{ij} \in \{0, 1\}$

Let $\mathbf{V}^* = \max_{i,j} \sum_{i \in I} \sum_{j \in J} \text{value}_{ij} x_{ij}$ be the optimal tutor-course assignment value found.

3.3 Stage 2: Workload Balance Optimization

This second stage ensures the assignment is the most equitable possible. Using the optimal tutor-course assignment value (V^*) established in Stage 1, this stage's objective is to minimize the unfairness of the workload by minimizing the range of tutor workload. The model searches through all possible assignments that achieve V^* and selects the single one that is the most balanced. This guarantees the final assignments are both of the highest possible quality and represent the fairest possible distribution of work.

Objective: Minimize $(\max_{i \in I} \sum_{j \in J} H_{ij} x_{ij} - \min_{i \in I} \sum_{j \in J} H_{ij} x_{ij})$

Subject to:

- **Tutor Workload Capacity:** Each tutor works maximum 10 hours a week

$$\sum_{j \in J} H_{ij} x_{ij} \leq 10 \qquad \forall i \in I$$

- **Course Covering:** Each course j gets its required demand

$$\sum_{i \in I} H_{ij} x_{ij} \geq d_j \qquad \forall j \in J$$

- **Tutor Course Minimum:** Each tutor is assigned at least 2 courses

$$\sum_{j \in J} x_{ij} \geq 2 \qquad \forall i \in I$$

- **Fixed Assignment Value:** Tutor-course assignment value must meet the standard set from stage 1 (V^*).

$$\sum_{i \in I} \sum_{j \in J} v_{ij} x_{ij} \geq V^*$$

- $x_{ij} \in \{0, 1\}$

4 Data Collection

The data used for this project was provided by the Student Academic Success Center (SASC) which consisted of two main components: tutor course preferences and utilization records from the first 11 weeks of this semester (Fall 2025).

The classes that each tutor is equipped or willing to teach to students is provided by the tutor themselves. Each tutor is required to provide at least two first choice courses, and are allowed to list any number of secondary-choice courses they are willing to tutor. Using this information, a matrix was produced to reflect preference in courses for each tutor. Each row in the matrix corresponded to a tutor, and each column represented a course that was listed to be supported by the center. Possible entries in the matrix took the values 0, 1, or 2 (Fig.1). Zero represents if the tutor did not list the course at all, 1 is if the course was listed as a secondary choice, and 2 represents if the course was listed as a first choice course.

Moving on to the utilization data, we exported all tutoring appointment information recorded during the first 11 weeks of the Fall 2025 semester from WCOnline, the platform used to schedule tutoring sessions and maintain appointment records. Each appointment record includes information on the course that the tutoring appointment was for, as well as the week or time it was requested. This was used to determine the volume or demand of each course for individual tutoring. Using the data, we were able to find weekly appointment counts for each course, which was used as the basis for estimating the course demand for our implementation later on. The total weekly demand was divided by the number of tutors that have been previously assigned to the course, resulting in an average per-tutor weekly utilization which was used in the hour cost matrix (Fig.2). We also record the total weekly demand for each course (Fig. 3) to quantify the center's expectations for course demand.

For any tutor who is not qualified to teach a course, the corresponding entry in the hour cost matrix was set to infinity. This step was crucial to ensure that any assignment attempt that was not a qualified pairing would violate the 10 hour maximum. This allows for tutor-course pair feasibility to be managed through the workload constraint.

	33121	33122	33141	33142	36200	36202
Tutor_1	0	0	0	0	0	1
Tutor_2	0	0	0	0	0	1
Tutor_3	0	0	1	0	0	0
Tutor_4	0	0	0	0	1	1
Tutor_5	0	0	0	0	2	2
Tutor_6	0	0	2	0	0	1
Tutor_7	0	0	0	0	0	0
Tutor_8	0	0	0	0	0	0
Tutor_9	0	0	0	0	0	0
Tutor_10	0	0	0	0	0	0
Tutor_11	1	2	0	0	0	0

Figure 1: Sample Preference (Value) Matrix

	33121	33122	33141	33142	36200	36202
Tutor_1	inf	inf	inf	inf	inf	0.06060607
Tutor_2	inf	inf	inf	inf	inf	0.06060607
Tutor_3	inf	inf	0.46464647	inf	inf	inf
Tutor_4	inf	inf	inf	inf	0.27272728	0.06060607
Tutor_5	inf	inf	inf	inf	0.27272728	0.06060607
Tutor_6	inf	inf	0.46464647	inf	inf	0.06060607
Tutor_7	inf	inf	inf	inf	inf	inf
Tutor_8	inf	inf	inf	inf	inf	inf
Tutor_9	inf	inf	inf	inf	inf	inf
Tutor_10	inf	inf	inf	inf	inf	inf
Tutor_11	2	0.54545455	inf	inf	inf	inf

Figure 2: Sample Hour Cost Matrix

Course	Hours
10301	1.18181818
15110	9.63636364
15112	7.63636364
15122	13.81818182
15150	2
15151	5.63636364
15210	0.90909091
15213	0.81818182
15251	1.63636364
15259	0.81818182
18100	4.54545455
18213	0.09090909
18220	1.27272727
18240	3.18181818
21090	5.27272727
21111	0.72727273
21112	0.63636364
21120	10.36363636
21122	1.36363636
21127	18.36363636
21128	3.90909091
21240	0.54545455
21241	8.18181818
21254	3.90909091
21256	2.27272727

Figure 3: Sample Total Course Demand

5 Implementation

5.1 Handling of Infeasible Assignments

In the input hour cost matrix, the infeasible pairings of a tutor and a course were denoted by infinity (inf). Because numerical solvers cannot process inf values, a program-

matic substitution was made during the model's implementation. Any inf was replaced with a large finite penalty of 100. Since this penalty is larger than the 10-hour working limit, the optimizer would never select these allowed assignments, effectively modeling the infeasibility.

5.2 Data Pre-processing and Feasibility Filtering

Prior to optimization, a data pre-processing step was implemented to eliminate sources of guaranteed infeasibility. This filtering logic was applied to both the tutor and course datasets based on two key constraints of the model.

- **Tutor Filtering:** The model's tutor course minimum constraint requires each tutor to be assigned to at least two courses. To enforce this, the initial value matrix was scanned for all tutors. Any tutor who indicated a valid preference for fewer than two courses was preemptively removed from the problem set, as they could not possibly satisfy this hard constraint.

- **Course Filtering:** A demand feasibility check was performed to identify and remove courses whose demand could not be met. In addition, any course that did not appear in the utilization dataset was removed from the value matrix to ensure that all modeled courses had defined, measurable demand. For each course, we first calculated its maximum potential supply. This value was determined by counting the total number of tutors who expressed a valid preference for that course and multiplying this count by the course's hours cost (the number of hours required from each assigned tutor). If this calculated potential supply was less than the course's total required demand, the course was removed from the dataset. This step was crucial as it prevents the model from attempting to find a solution for a provably infeasible state where the course covering constraint cannot be met.

This pre-filtering process ensured that the inputs to the optimization models were valid and feasible with respect to these two fundamental constraints.

5.3 Model Execution

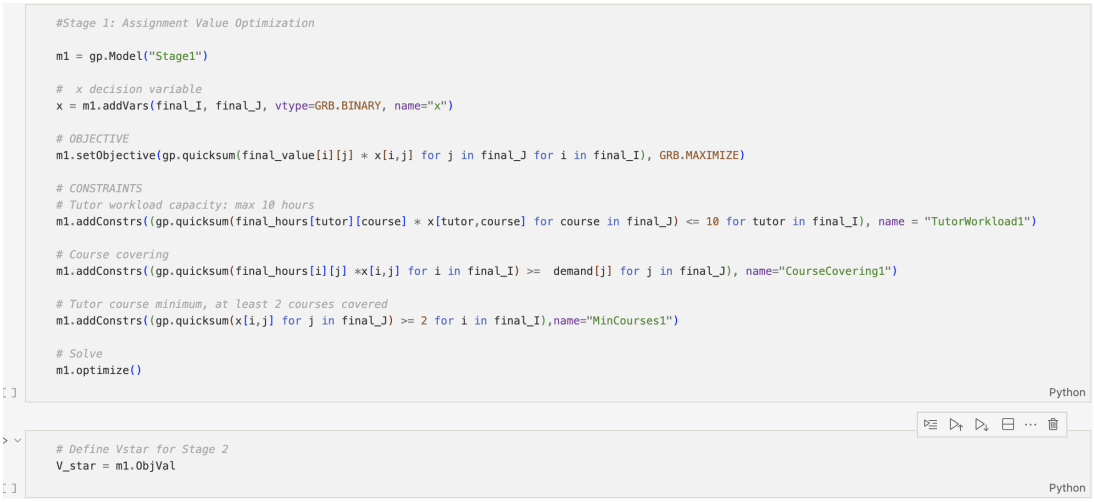
The optimization models were implemented in Python using Gurobi. All data matrices that were derived earlier were imported from CSV spreadsheets and converted into NumPy

arrays before we began implementation. The first stage of the model was solved to obtain an optimal assignment value V^*, which represented the maximum feasible preference score for all tutor-course assignments. Once this value was found, the second execution consisted of an additional constraint being considered. This constraint required the assignment value to meet or exceed V^* to ensure that the balancing of workloads did not come with the cost of assignment quality. All optimizations were performed on a standard laptop using the default Gurobi solver settings.

6 Results

6.1 Model Optimization

Using Gurobi Optimizer with Python, we produced the following code:



```python
#Stage 1: Assignment Value Optimization

m1 = gp.Model("Stage1")

#  x decision variable
x = m1.addVars(final_I, final_J, vtype=GRB.BINARY, name="x")

# OBJECTIVE
m1.setObjective(gp.quicksum(final_value[i][j] * x[i,j] for j in final_J for i in final_I), GRB.MAXIMIZE)

# CONSTRAINTS
# Tutor workload capacity: max 10 hours
m1.addConstrs((gp.quicksum(final_hours[tutor][course] * x[tutor,course] for course in final_J) <= 10 for tutor in final_I), name = "TutorWorkload1")

# Course covering
m1.addConstrs((gp.quicksum(final_hours[i][j] *x[i,j] for i in final_I) >=  demand[j] for j in final_J), name="CourseCovering1")

# Tutor course minimum, at least 2 courses covered
m1.addConstrs((gp.quicksum(x[i,j] for j in final_J) >= 2 for i in final_I),name="MinCourses1")

# Solve
m1.optimize()
```

Python

```python
# Define Vstar for Stage 2
V_star = m1.ObjVal
```

Python

Figure 4: Stage 1 - Assignment Value Optimization



```python
#  Stage 2: Workload Balance Optimization
m2 = gp.Model("Stage2")

#  x decision variable
x = m2.addVars(final_I, final_J, vtype=GRB.BINARY, name="x")

# --- 1. Calculate Total Hours for Each Tutor ---
total_hours = m2.addVars(I, name="total_hours_per_tutor", lb=0.0)
for i in final_I:
    m2.addConstr(
        total_hours[i] == gp.quicksum(final_hours[i][j] * x[i,j] for j in final_J),
        name=f"calc_total_hours_{i}"
    )

# Find the Overall Max and Min
max_hours_var = m2.addVar(name="max_workload", lb=0.0)
min_hours_var = m2.addVar(name="min_workload", lb=0.0)

m2.addGenConstrMax(max_hours_var, total_hours, name="find_max_hours")
m2.addGenConstrMin(min_hours_var, total_hours, name="find_min_hours")

# Define and Set the Objective

# Create a variable to represent the difference
workload_spread = m2.addVar(name="workload_spread", lb=0.0)

# Add a constraint that defines this variable
m2.addConstr(
    workload_spread == max_hours_var - min_hours_var,
    name="calc_workload_spread"
)

# OBJECTIVE
m2.setObjective(workload_spread, GRB.MINIMIZE)


# CONSTRAINTs

# Tutor workload capacity: max 10 hours
m2.addConstrs((gp.quicksum(final_hours[tutor][course] * x[tutor,course] for course in final_J) <= 10 for tutor in final_I), name = "TutorWorkload1")

# Course covering
m2.addConstrs((gp.quicksum(final_hours[i][j] *x[i,j] for i in final_I) >=  demand[j] for j in final_J), name="CourseCovering1")

# Tutor course minimum, at least 2 courses covered
m2.addConstrs((gp.quicksum(x[i,j] for j in final_J) >= 2 for i in final_I),name="MinCourses1")

# fixed value
m2.addConstr(gp.quicksum(final_value[i][j] * x[i,j] for j in final_J for i in final_I) >= V_star, name="FixedValue")

#Solve
m2.optimize()
```

Figure 5: Stage 2 - Workload Balance Optimization

Stage 1 maximized assignment quality while keeping the maximum workload for any

single tutor at approximately 9.81 hours. Stage 2 then searched among all assignments that achieve this same high value and selected the one that minimizes the workload range. The resulting solution assigns a maximum of 9.72 hours and a minimum of 0.36 hours per tutor, yielding a range of 9.36 hours. This indicates that, under the current constraints, preserving maximum assignment value leads to a uneven distribution of work hours. While the model successfully meets course coverage requirements and maintains high assignment quality, the resulting imbalance suggests that additional constraints may be needed in future iterations to better support fairness in workload among tutors.

6.2 Tutor Assignments

To visualize our results, we produced the heatmaps shown in figures 6 and 7.

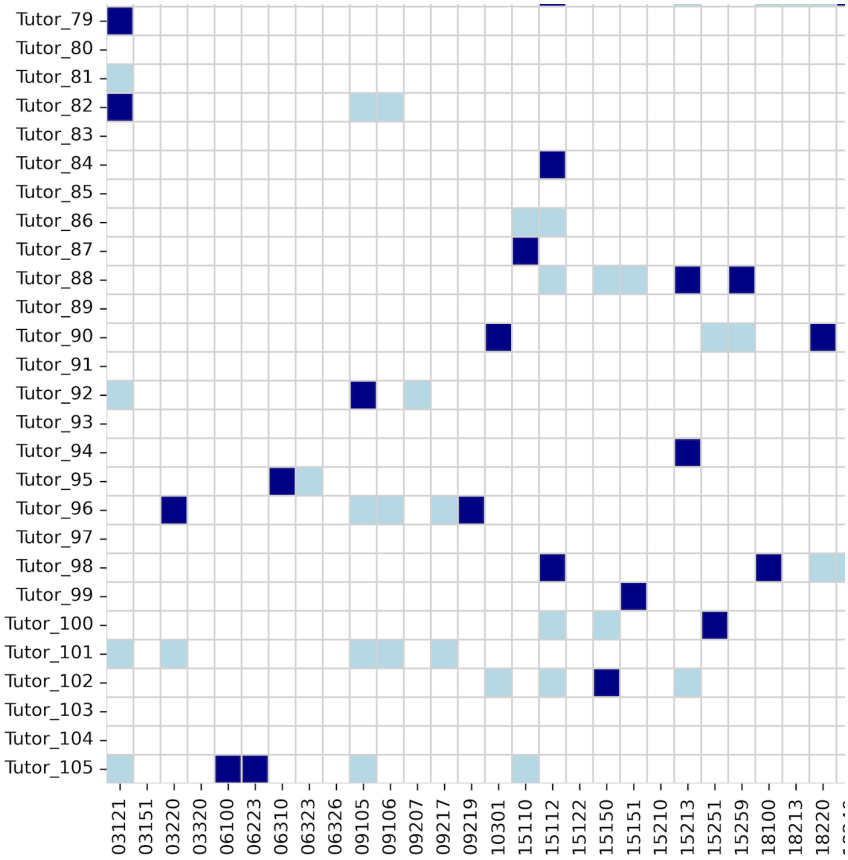


Figure 6: Optimal Tutor Assignments Heatmap (Detailed View)

Figure 6 is a zoomed-in heatmap of the tutor assignments. The optimized model visualizes the suitability of each tutor for each course based on self-reported preference data. As mentioned above: 0 indicates that the tutor did not list the course, 1 indicates that

the course was listed as a secondary option, and 2 indicates that the course was listed as a primary choice. In the heatmap, dark blue represents a value of 2, light blue indicates value of 1, and white is value zero. For example, optimally, Tutor_96 prefers to tutor and is the best fit to tutor courses 03220 and 09219. This corresponds to Genetics and Modern Organic Chemistry 1 respectively.

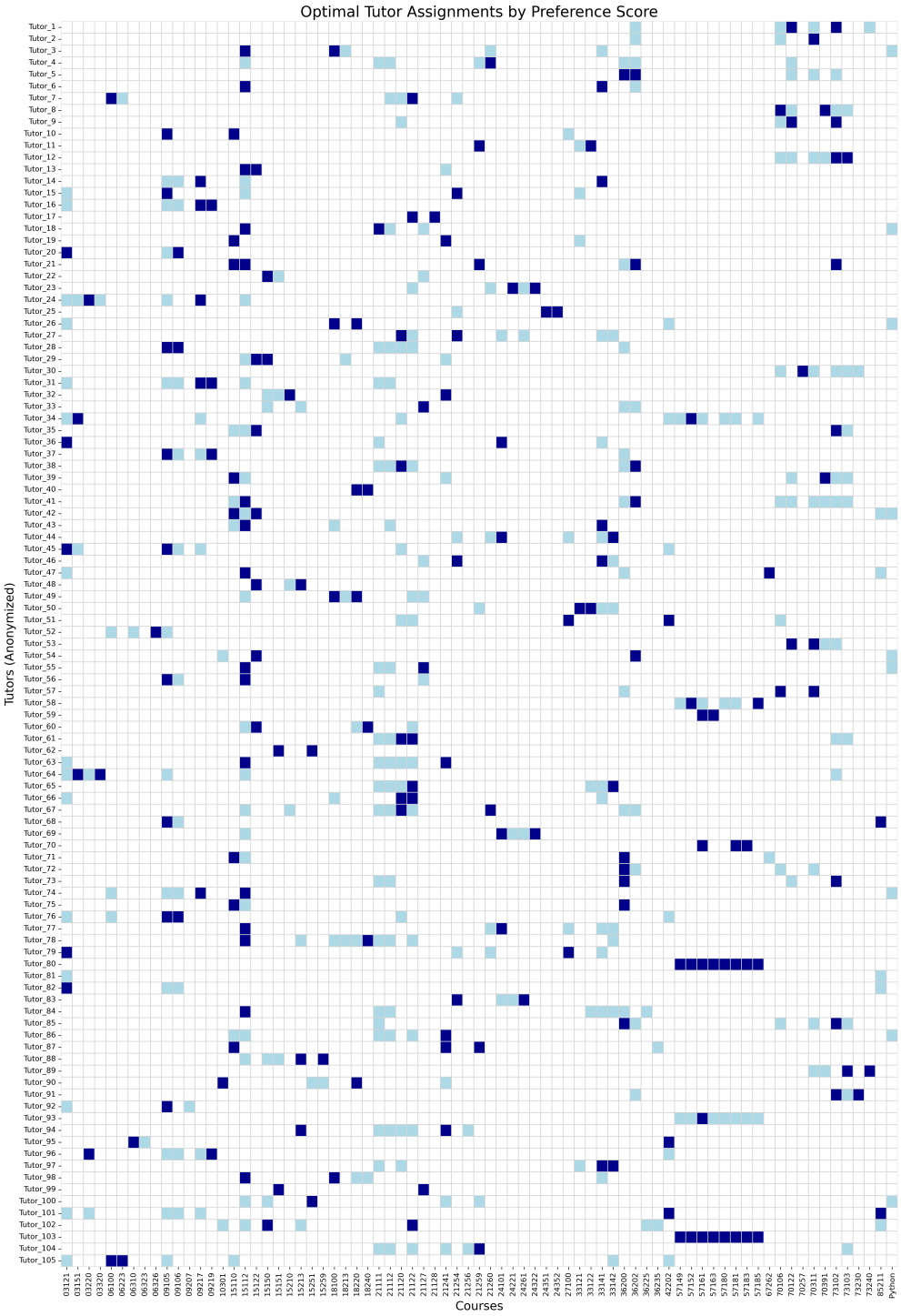


Figure 7: Optimal Tutor Assignments Full Heatmap

Comparing Figure 7 with the original preference matrix, we see that a majority of tutors would have their first-choice courses. The figure also reveals a sparse assignment structure. Most tutors are assigned only a small number of courses, reflecting SASC's tutoring specialization. Similarly, certain courses display high assignment density, while others remain lightly staffed. From the heatmap, courses such as 15112 (Fundamentals of Programming) and 21120 (Differential and Integral Calculus) have a high density while 24351 (Dynamics) has a low density. It is important to note that this is a good representation of demand since courses like 15112 are larger and tend to have a higher workload.

6.3 Optimality Plots

To visualize if we met our two goals of better matching demand and reducing the number of underutilized tutors, we created histograms based on demand and the number of hours worked. These plots are shown in figures 8 and 9. The data comes from the optimal assignment summary CSV. Each tutor/course pair is assigned a value of (-1) or some other number. The negative one represents the tutor not being assigned to that specific course, and other values represent weekly demand and hour cost per tutor for the courses they are assigned to.

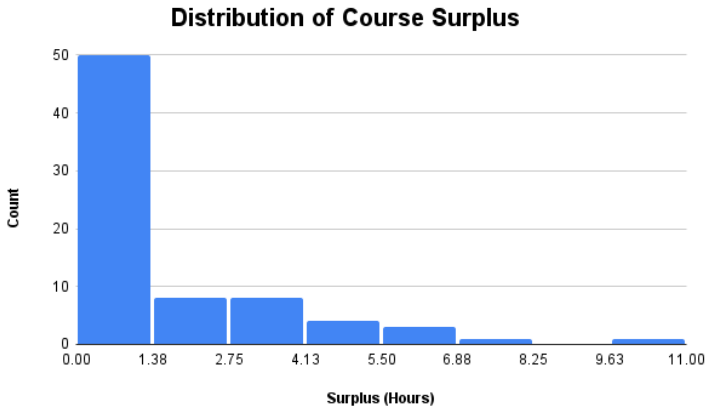


Figure 8: Distribution of Course Surplus

Figure 8 shows the course surplus distribution. These values were calculated by subtracting the actual demand from the total calculated demand in the optimal assignment data file. The histogram is right-skewed and most of the course surplus is zero. This indicates that a majority of the courses have an optimal tutor/course matching.

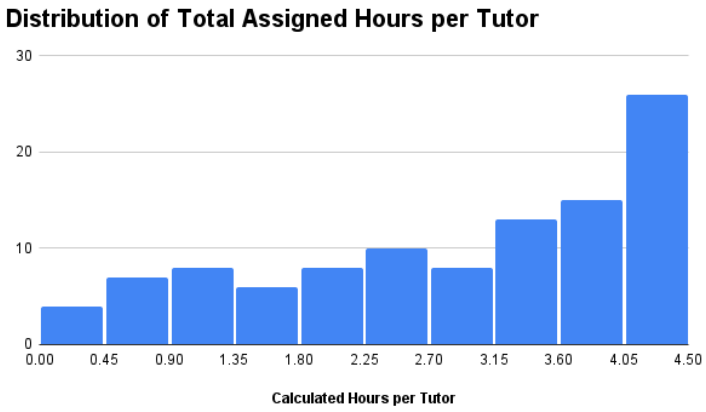


Figure 9: Distribution of Total Assigned Hours per Tutor

Figure 9 shows how the workload is spread across the tutors. Ideally, all tutors would have a similar value for total hours. The histogram shows a wide-spread variability in workload. Although many tutors cluster around mid-range values, there are some tutors who receive significantly more hours at around 4.5 hours. This indicates that there is still under- and over-utilization of some tutors.

6.4 Interpretation of Final Results

Although individual tutor workloads vary in Stage 2, the model successfully meets its primary objective of optimizing assignment quality while keeping all workloads below the 10-hour weekly threshold. The heatmap results confirm that most tutors are assigned to their first-choice courses, reflecting a strong alignment between tutor expertise and course needs. This is also more beneficial for students, since the tutor would more than likely be in that specific major they are tutoring courses for. The histograms indicate that our model allocates tutors efficiently to certain courses. However, the workload between tutors is not evenly distributed. Classes with high enrollment and greater tutoring demand receive a higher concentration of tutor assignments, while lower-demand courses are assigned fewer tutors. These results indicate that the optimization model captures realistic tutoring patterns. We are able to achieve the first goal of better matching course demands. Our second goal to optimize workload balance is not fully achieved. This can be improved with more data from previous years and implementing more equity constraints related to workload in the model.

7 Limitations and Future Work

Although the current model successfully finds an optimal assignment, this optimality is highly dependent on the assumptions we made regarding demand, tutor availability, and role constraints.

The first limitation involves individual tutor availability, as our model assumes all tutors share the same maximum capacity of 10 hours for individual tutoring. However, this does not consider the individual availability limits, or the fact that some tutors may be limited to a lower maximum limit due to other responsibilities. Another limitation is only considering individual tutoring sessions into our model. SASC provides other modes of tutoring, one being drop-in sessions for specific courses. This also contributes to each tutor's weekly hours. This would result in a potential over-assignment or under-assignment of total hours including the drop-in sessions that tutors may work.

Finally, our input metrics, including the hour-cost matrix and demand estimates, were derived from the data for this semester (September through mid-November). Because these values may differ from future demand patterns, the resulting assignments may not fully meet student needs or may lead to inefficient tutor utilization in future semesters.

Thus, to address this potential inaccuracy, future work could incorporate more detailed tutor availability information, including drop-in tutoring hours. These hours could be treated as fixed prerequisites or introduced as an additional variable to be optimized along with individual assignments. Because tutor qualifications and course preferences change each semester, another important extension would be to develop an administrative interface that makes the model directly usable. Such an interface would allow SASC staff to input current tutor data at the start of each semester and immediately view optimized tutor–course assignments. The system could also flag courses with too many or too few qualified tutors, informing the hiring process rather than only producing assignments for tutors.

8 Acknowledgements

We would like to thank Wesley Hood, the Assistant Director for Learning Support at SASC, for providing data for our project.

References

[1] Data from the Student Academic Success Center

A Appendix: Code

The code in this appendix includes the pre-processing steps required to prepare the input data, the procedures for running the model, and the generation of the final heatmap output. The code takes as inputs the value matrix, hour cost matrix, weekly demand, and per-tutor weekly demand.

```
import gurobipy as gp
from gurobipy import GRB, quicksum
import pandas as pd
import numpy as np
import copy
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.colors import ListedColormap, BoundaryNorm

INF_REPLACEMENT = 100

# Import data
values_df = pd.read_csv('Data/preference_matrix.csv', index_col=0)
demand_df = pd.read_csv('Data/Total Demand.csv')
hours_df = pd.read_csv('Data/tutor_utilization_matrix.csv', index_col=0)
per_tutor_demand_df = pd.read_csv('Data/Per Tutor Demand.csv')

# replace inf with 100
hours_df = hours_df.where(np.isfinite(hours_df), INF_REPLACEMENT)


# ------------------ DEFINE VARIABLES ------------------

J = list(values_df.columns)        # courses
J = [str(j).zfill(5) for j in J]   # standardize the course names

I = list(values_df.index)        # tutors

# Define dictionaries and standardize Course names
demand = dict(zip(demand_df["Course"], demand_df["hours"]))
demand = {str(k).zfill(5): v for k,v in demand.items()}

per_tutor_demand = dict(zip(per_tutor_demand_df["Course"],
    per_tutor_demand_df["hours"]))
per_tutor_demand = {str(k).zfill(5): v for k,v in per_tutor_demand.items()}

value = values_df.to_dict(orient="index")
value = {
    andrewid: {str(course).zfill(5): v for course, v in inner_dict.items()}
```

```
    for andrewid, inner_dict in value.items()
}

# create hours dictionary and pad course names with zero + convert to string
hours = hours_df.to_dict(orient="index")  # more parsed df like hours[i][j] [course col: tutor col: row hours]
hours = {
    andrewid: {str(course).zfill(5): v for course, v in inner_dict.items()}
    for andrewid, inner_dict in hours.items()
}


# ------------------  Pre-processing and Feasibility Filtering ------------------

# Count tutors with preference > 0 for each course
tutor_counts = {}

# For each course in the master list J
for j in J:
    count_for_this_course = 0

    # Loop through every tutor in the master list I
    for i in I:

        # Check the preference score: value[tutor][course]
        try:
            if value[i][j] > 0:
                count_for_this_course += 1
        except KeyError:
            # This handles tutor or course missing values
            pass

    tutor_counts[j] = count_for_this_course


# Calculate supply and check for shortfalls
analysis_results = {}
shortfall_courses = []

# Loop through the master course list J
for j in J:

    # Get the demand for this course from the demand dict
    per_tutor_hours = per_tutor_demand.get(j, 0)
    demand_needed = demand.get(j, 0)

```

```
85      # Get the count of available tutors for this course from above
86      count = tutor_counts[j]
87
88      # Calculate "Potential Supply" (Tutor Count * Weekly Demand)
89      potential_supply_tutor_hours = count * per_tutor_hours
90
91      # Check if the calculated value is less than the demand
92      # We check if demand_hours is > 0 to avoid (0 < 0)
93      if demand_needed > 0 and potential_supply_tutor_hours < demand_needed:
94          shortfall_courses.append(j)
95
96
97  # Create new, independent copies to avoid changing your originals
98  final_value = copy.deepcopy(value)
99  final_hours = copy.deepcopy(hours)
100
101 # Remove indicated course(s) found above
102 for tutor_name in I:
103
104     # Loop through the list of courses to remove
105     for course_to_remove in shortfall_courses:
106
107         # Remove the course from both new dictionaries
108         if tutor_name in final_value:
109             final_value[tutor_name].pop(course_to_remove, None)
110
111         if tutor_name in final_hours:
112             final_hours[tutor_name].pop(course_to_remove, None)
113
114 # update set of courses
115 final_J = [course for course in J if course not in set(shortfall_courses)]
116
117 # Count number of course preferences and check for too few course interests
118 tutors_with_few_prefs = []
119
120 for i in I:
121
122     # Get the tutor's preference dictionary.
123     tutor_prefs = final_value.get(i, {})
124
125     # Count the number of "valid" preferences (non zero preferences)
126     valid_pref_count = sum(1 for pref_score in tutor_prefs.values() if
        pref_score != 0)
127
128     # Check if the count is less than 2
129     if valid_pref_count < 2:
130         tutors_with_few_prefs.append(i)
131
```

```
# removed indicated tutors from both new dictionaries
for tutor_to_remove in tutors_with_few_prefs:
    final_value.pop(tutor_to_remove, None)
    final_hours.pop(tutor_to_remove, None)

# update set of tutors
final_I =  [tutor for tutor in I if tutor not in set(tutors_with_few_prefs)]



# ------------------  Stage 1: Assignment Value Optimization ------------------
# Model 1 for stage 1
m1 = gp.Model("Stage1")

#  x decision variable
x = m1.addVars(final_I, final_J, vtype=GRB.BINARY, name="x")

# OBJECTIVE
m1.setObjective(gp.quicksum(final_value[i][j] * x[i,j] for j in final_J for i in final_I), GRB.MAXIMIZE)

# CONSTRAINTS
# Tutor workload capacity: max 10 hours
m1.addConstrs((gp.quicksum(final_hours[tutor][course] * x[tutor,course] for course in final_J) <= 10 for tutor in final_I), name = "TutorWorkload1")

# Course covering
m1.addConstrs((gp.quicksum(final_hours[i][j] *x[i,j] for i in final_I) >= demand[j] for j in final_J), name="CourseCovering1")

# Tutor course minimum, at least 2 courses covered
m1.addConstrs((gp.quicksum(x[i,j] for j in final_J) >= 2 for i in final_I), name="MinCourses1")

# Solve
m1.optimize()

# Define Vstar for Stage 2
V_star = m1.ObjVal


# ------------------  Stage 2: Workload Balance Optimization ------------------
# Model 2 for Stage 2
m2 = gp.Model("Stage2")
```

```
172
173 #  x decision variable
174 x = m2.addVars(final_I, final_J, vtype=GRB.BINARY, name="x")
175
176 # --- 1. Calculate Total Hours for Each Tutor ---
177 total_hours = m2.addVars(I, name="total_hours_per_tutor", lb=0.0)
178 for i in final_I:
179     m2.addConstr(
180         total_hours[i] == gp.quicksum(final_hours[i][j] * x[i,j] for j in
        final_J),
181         name=f"calc_total_hours_{i}"
182     )
183
184 # Find the Overall Max and Min
185 max_hours_var = m2.addVar(name="max_workload", lb=0.0)
186 min_hours_var = m2.addVar(name="min_workload", lb=0.0)
187
188 m2.addGenConstrMax(max_hours_var, total_hours, name="find_max_hours")
189 m2.addGenConstrMin(min_hours_var, total_hours, name="find_min_hours")
190
191 # Define and Set the Objective
192
193 # Create a variable to represent the difference
194 workload_spread = m2.addVar(name="workload_spread", lb=0.0)
195
196 # Add a constraint that defines this variable
197 m2.addConstr(
198     workload_spread == max_hours_var - min_hours_var,
199     name="calc_workload_spread"
200 )
201
202 # OBJECTIVE
203 m2.setObjective(workload_spread, GRB.MINIMIZE)
204
205
206 # CONSTRAINTs
207
208 # Tutor workload capacity: max 10 hours
209 m2.addConstrs((gp.quicksum(final_hours[tutor][course] * x[tutor,course] for
        course in final_J) <= 10 for tutor in final_I), name = "
       TutorWorkload1")
210
211 # Course covering
212 m2.addConstrs((gp.quicksum(final_hours[i][j] *x[i,j] for i in final_I) >=
       demand[j] for j in final_J), name="CourseCovering1")
213
214 # Tutor course minimum, at least 2 courses covered
```

```
m2.addConstrs((gp.quicksum(x[i,j] for j in final_J) >= 2 for i in final_I),
    name="MinCourses1")

# fixed value
m2.addConstr(gp.quicksum(final_value[i][j] * x[i,j] for j in final_J for i
    in final_I) >= V_star, name="FixedValue")

#Solve
m2.optimize()

# %%
# ------------------  Create Heatmap Result ------------------

if m2.Status == GRB.OPTIMAL:

    print("Optimal solution found. Anonymizing and saving all outputs...")

    # --- Create Anonymization Key ---

    # Create a new list of anonymous IDs
    anonymous_tutor_ids = [f"Tutor_{i+1}" for i in range(len(final_I))]

    # Create the mapping dictionary (Original → Anonymous)
    # This is the "key"
    id_mapping = dict(zip(final_I, anonymous_tutor_ids))

    # Save the key to a CSV file
    try:
        mapping_df = pd.DataFrame(list(id_mapping.items()),
                                  columns=['Original_ID', 'Anonymous_ID'])
        mapping_df.to_csv("tutor_anonymization_key.csv", index=False)
        print("--- Saved anonymization key to tutor_anonymization_key.csv
    ---")
    except Exception as e:
        print(f"--- Error saving anonymization key: {e} ---")


    # --- Create Anonymized CSV Summary ---

    # Create DataFrame, initializing with -1.0
    results_df = pd.DataFrame(index=final_I, columns=final_J, data=-1.0)

    # Loop solution (using original IDs)
    for i in final_I:
        for j in final_J:
            if x[i, j].X > 0.5:
                results_df.loc[i, j] = final_hours[i][j]

```

```
    # Add 'Tutor Total Hours' column
    results_df['Tutor Total Hours'] = results_df.apply(
        lambda row: row[row >= 0].sum(), axis=1
    )

    # --- Apply Anonymization to the Index ---
    # Get the original tutor index
    original_tutor_index = list(results_df.index)
    # Map them to the new anonymous IDs
    anonymous_index = [id_mapping.get(i) for i in original_tutor_index]

    # Add summary rows
    results_df.loc['Total Calculated Demand'] = results_df.apply(
        lambda col: col[col >= 0].sum(), axis=0
    )
    results_df.loc['Actual Needed Demand'] = pd.Series(demand)
    results_df.loc['Shortfall/Surplus'] = \
        results_df.loc['Total Calculated Demand'] - results_df.loc['Actual
    Needed Demand']
    results_df = results_df.fillna('')

    # Set the new anonymized index + the summary rows
    summary_row_names = ['Total Calculated Demand', 'Actual Needed Demand
    ', 'Shortfall/Surplus']
    results_df.index = anonymous_index + summary_row_names

    # Save the anonymized CSV with summary numbers
    try:
        results_df.to_csv("anonymized_assignments_summary.csv")
        print("--- Saved anonymized summary CSV to
    anonymized_assignments_summary.csv ---")
    except Exception as e:
        print(f"---Error saving summary CSV: {e} ---")


    # --- Create Anonymized Heatmap ---

    # Create the DataFrame for the plot (using original IDs first)
    plot_df = pd.DataFrame(index=final_I, columns=final_J, data=-1.0)
    for i in final_I:
        for j in final_J:
            if x[i, j].X > 0.5:
                score = final_value.get(i, {}).get(j, 0)
                plot_df.loc[i, j] = score

    # --- Apply Anonymization to the Plot DataFrame ---
    plot_df.index = [id_mapping.get(i) for i in plot_df.index]

```

```
    # Pre-process for plotting (0=white, 1=lightblue, 2=darkblue)
    def map_values_for_plot(x):
        if x == 1: return 1
        if x == 2: return 2
        return 0
    final_plot_df = plot_df.applymap(map_values_for_plot)

    # Define colormap and boundaries
    colors = ['white', 'lightblue', 'darkblue']
    cmap = ListedColormap(colors)
    boundaries = [-0.5, 0.5, 1.5, 2.5]
    norm = BoundaryNorm(boundaries, cmap.N)

    # Generate and save the heatmap
    print("Generating anonymized heatmap image...")
    plt.figure(figsize=(20, 30)) # Large size for 100x60
    sns.heatmap(
        final_plot_df,
        cmap=cmap, norm=norm, cbar=False,
        linewidths=0.5, linecolor='lightgray'
    )
    plt.title('Optimal Tutor Assignments by Preference Score', fontsize=
    20)
    plt.xlabel('Courses', fontsize=16)
    plt.ylabel('Tutors (Anonymized)', fontsize=16) # Updated y-label

    # Saves the heatmap as a file
    try:
        plt.savefig(αnonymized_assignment_heatmap.png', bbox_inches='tight
    ', dpi=300)
        print("--- Heatmap saved to anonymized_assignment_heatmap.png ---")
    except Exception as e:
        print(f"--- Error saving heatmap: {e} ---")

else:
    print("No optimal solution found. No files were saved.")
```