Final Report Data Analysis Lab

Empirical analysis of a many-goods-to-one-agent allocation problem

By
Richard Kempert, 11915893
University of Innsbruck
Digital Science Center
Course: Data Analysis Lab
Instructors: Prof Dr Adam Jatowt & Dr Alexander Kupfer

18th July 2022

Relevance and Description of the Implemented Allocation Mechanisms

One of the key concerns of economics is to efficiently allocate and distribute resources. Probably the most know example of such an allocation mechanism is the price-based market where supply and demand regulate the distribution of resources. Beyond price-based allocations, we face several non-price-based allocation problems, e.g., organ donation (deciding who receives an organ) or student houses (who gets which room/flat). This project engages also with one of these allocation problems, namely the many-goods-to-one-agent problem. This problem is characterized by a finite set of agents who want to allocate a finite set of indivisible goods, whereby there are more goods than agents and each agent can allocate several goods.

The problem of allocating a finite set of indivisible goods to a finite set of agents is known and has been scientifically analyzed. Especially, much research has been done from a mathematical perspective when agents can select only one good. Svensson (1999) proofed mathematically that the Serial Dictatorship (SD) mechanism is an efficient allocation of an "one-sided" market. The SD is quite simple. The first step is to permutate the order of agents. Second, each agent picks a single good and the picked good is removed from the set of goods, whereas the first agent from the random order starts then the second agent and so on. Although quite simple, the SD has remarkable properties. It produces pareto optimal allocations, thus no changes are possible without making at least one agent worse off. Moreover, it is strategy proof meaning that telling one's true preferences is a dominant strategy and thus, it is always the best option no matter what other agents do (Svensson, 1999). Furthermore, it is *ex antes* envyfree since every agent has the same probability to select their most preferred good (Lien et al., 2017). But these properties are only valid in this simple case (Manea, 2007).

Nevertheless, the SD can be applied for the more complex allocation problems where each agent can allocate several goods. It changes in the sense that instead of picking one good, each agent picks *x* goods ($x = \frac{g\,(number\ of\ goods)}{a\,(number\ of\ agents)}$). If *x* is not an integer, the first agents in the random order are rounded up and the last agents are rounded down. This mechanism would at least be strategy proof and *ex antes* envyfree (Manea, 2007). A second possible mechanism could be to do several rounds of picking. This adaption of SD is called 'round-robin' item allocation. Although this allocation does not yield pareto optimal allocations (Manea, 2007), it is *ex post* envyfree up to one good. This means that every agent values his_her bundle at least as much as other agents' bundles if at most one good is removed from the other bundle, which is weaker than envyfree (Caragiannis et al., 2016). Following Caragainnis et al. (2016), a third mechanism is the application of 'Maximum Nash Welfare' which leads to surprisingly

fair results. However, it requires that the agents not only have strict preferences but also assign an explicit utility value to the individual goods, which is not the case with the sample data of this project.

Besides all these mathematical analyses of this allocation problem, empirical examinations are rare. To the best of my knowledge, only Abdulkadroğlu et al. (2017) investigated the results of applying SD in a real case (students choosing schools). Due to the increased complexity of many-objects-to-one-agent-allocations, further research is needed (Manea, 2007). Especially, the lack of empirical analysis exploring the results and differences between allocation mechanisms in real cases could provide further insights and new perspectives. Thus, the aim of this research project is to implement and compute different allocation mechanisms and compare the results based on selected criteria.

The setting of the allocation problem can be described as having a finite set of indivisible goods and a finite set of agents, whereby there are more goods than agents and each agent can have several goods. To explore this allocation problem, several allocation mechanisms are programmatically implemented, and the results are empirically compared. I have selected allocation mechanisms that are relatively easy to understand and therefore, feasible to implement in practice. The used data are real life data from an organization that allocates the goods (=course seats) to agents (=trainers). Each trainer gives preferences over courses beforehand, which means they have an ordered ranking of all the courses they wish to allocation. The purpose of the allocation mechanisms is – based on the trainers' preferences – to 'decide' which trainer gets which course seat.

The different allocation mechanisms that are implemented in this project can be divided into three categories: (1) a classical serial dictatorship, (2) round-robin allocations, and (3) random allocations from the "course side". The steps of all implemented allocation mechanisms are visualized below in Figure 1-3. Figure 1 illustrates the steps of the classical SD. The first step is to permutate the order of trainers. The first trainer of this order allocates all his_her preferred courses if they are available. After a course seat has been assigned, it is removed from the set of courses. Afterwards, the second trainer does the same and so on.

Figure 1

Implementation of the serial dictatorship (own illustration)

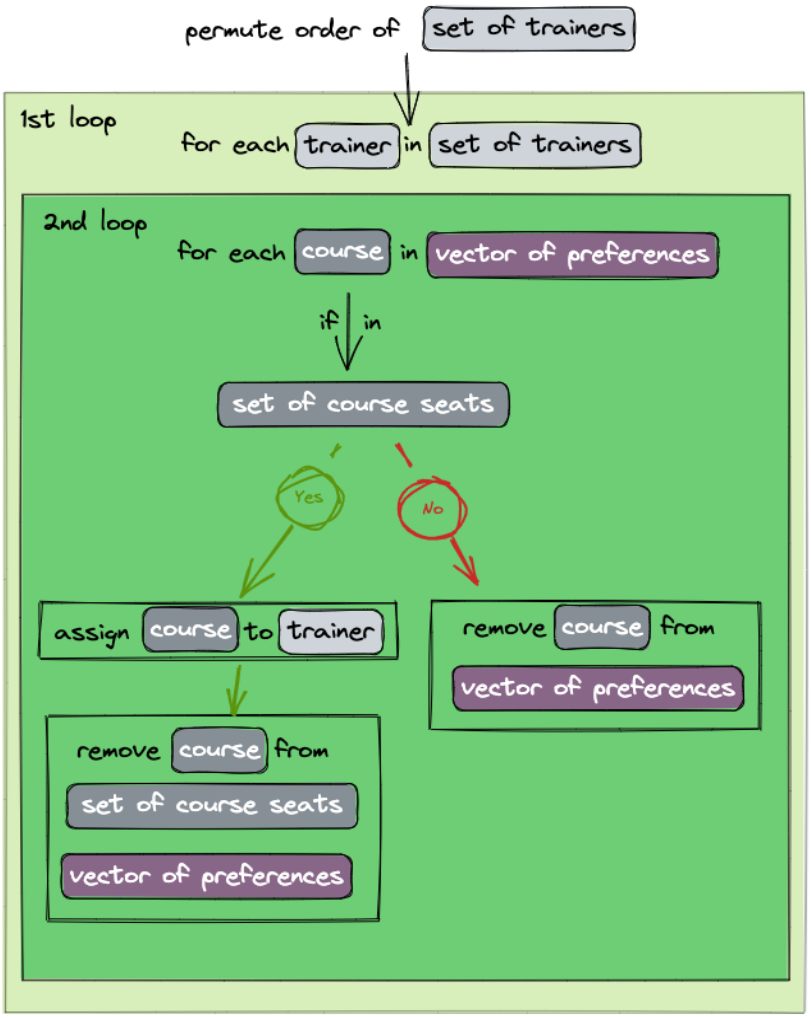


Figure 2 presents the three versions of the implemented round-robin allocations. All versions start in the same way as SD, with a random order of all trainers. In the second step, the first trainer allocates his_her most preferred course that is in the set of courses. This course is then removed from his_her vector of preferences and from the set of courses. The second trainer does the same and so on. When all trainers had a turn to allocated one course seat, the first trainer starts again with choosing his_her most preferred course that is available and the whole process starts all over again. This process ends either if no more course seats are available or if no trainer has preferences over any course seats that are available. The second version differs from version 1 in the number of courses seats each trainer can allocate in each picking round, e.g., instead of selecting only one course, each trainer allocates his_her three most preferred courses from the set of courses. Version 3 re-permutates the picking order of trainers after each allocation round. Thus, the first trainer of each picking round changes randomly and does not stay the same by purpose.

Figure 2

The three implemented versions of round-robin allocations (own illustration)

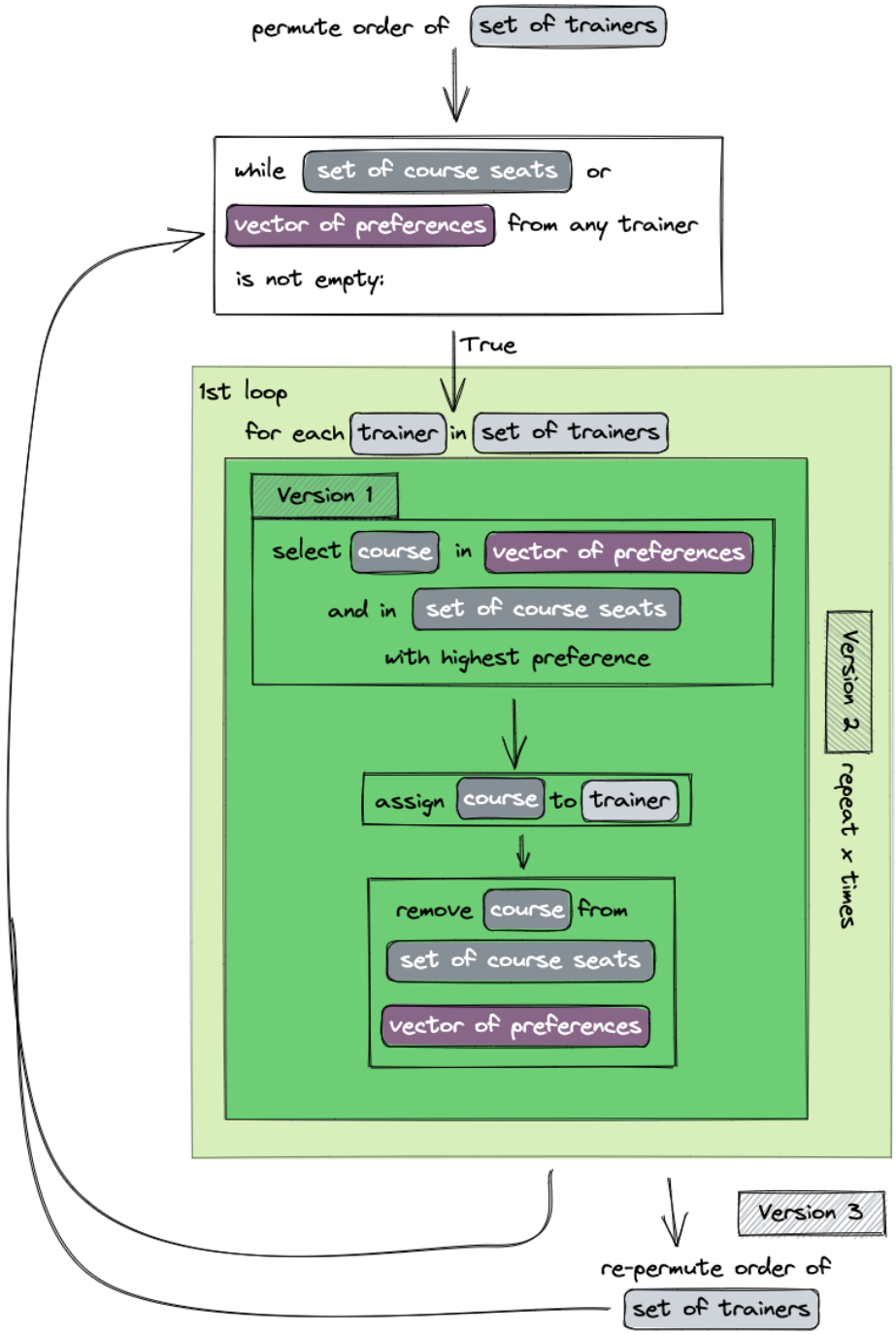
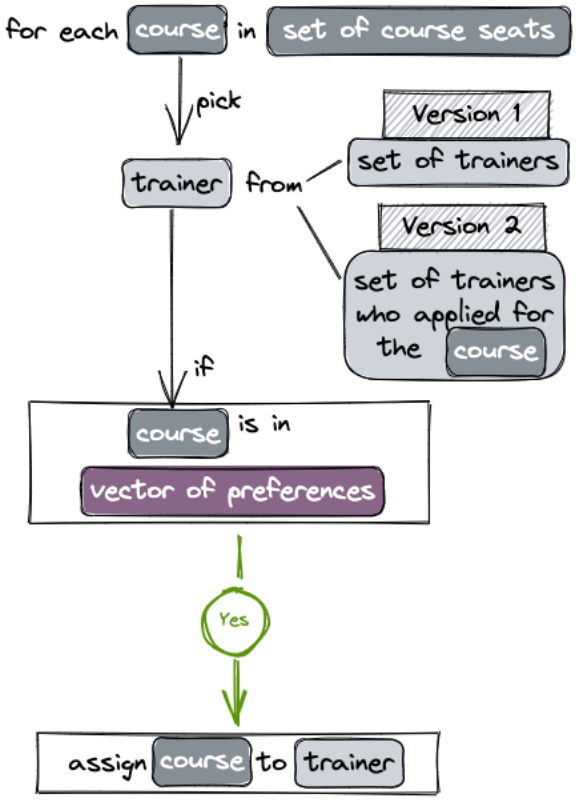


Figure 3 displays the random allocations, and hence, the fifth and sixth allocation mechanism. Whereas SD and round-robin iterate over all trainers, the two versions of the "course side" iterate over all courses and assign as much trainers as required. The two versions differ in the regard that version 1 assigns randomly from the set of trainers and version 2 assigns trainers from the set of trainers that have preferences over the respective course.

Figure 3

Allocation mechanism from the "course side" (own illustration)



Research Hypothesis

Since the sample data do not contain information about the utility of each course seat for each trainer, the question arises how we can judge if the allocation is "efficient". Out of the trainers' perspective, it is not clear what characterizes an efficient allocation. Since there is no information about the utility of each course, I assume that the main utility of a course is to allocate one, so each course has the same utility. Since the trainers prefer different numbers of courses and it is not the goal that all trainers allocate the same amount of course seats, the absolute number of courses is not a good expression of the utility. Instead, the relative ratio between preferred courses and allocated courses seems to be a useful variable to consider the efficiency of an allocation. Inspired by the active scientific discussion about fairness of allocating indivisible goods (see Biswas & Barman, 2019), I argue that an efficient allocation is as fair as possible (low standard deviation) and at the same time all trainers can allocate as many preferred courses seats as possible (high mean). Based on this consideration, the coefficient of variation (CV), measuring the spread of values relative to the mean of the dataset, seems to be a suitable measurement for the efficiency of an allocation. From the course side,

it seems reasonable to argue that an efficient allocation is one that leaves as few open course seats as possible, meaning a high ratio of assigned course seats to the total number of course seats. Thus, I derive the following hypothesis:

H_0 1: The ratio of assigned course seats does not differ between different allocation mechanisms.

H_1 1: The ratio of assigned course seats does differ between different allocation mechanisms.

H_0 2: The coefficient of variation of application success rate does not differ between different allocation mechanisms.

H_1 2: The coefficient of variation of application success rate does differ between different allocation mechanisms.

Methods and Workflow

To reach the aim of this research project and to test the hypothesis, the following workflow was conducted: (1) getting access to useful raw data, (2) preparing the data in a way that enables an implementation of the different allocation mechanisms, (3) implementing the different mechanisms, (4) cleaning and processing the data in a way that yields the application success rate respective the coefficient of variation and the ratio of filled course seats, and lastly (5) analyzing the cleaned data. Figure 4 summarizes the whole workflow, followed by a detailed description of the steps.

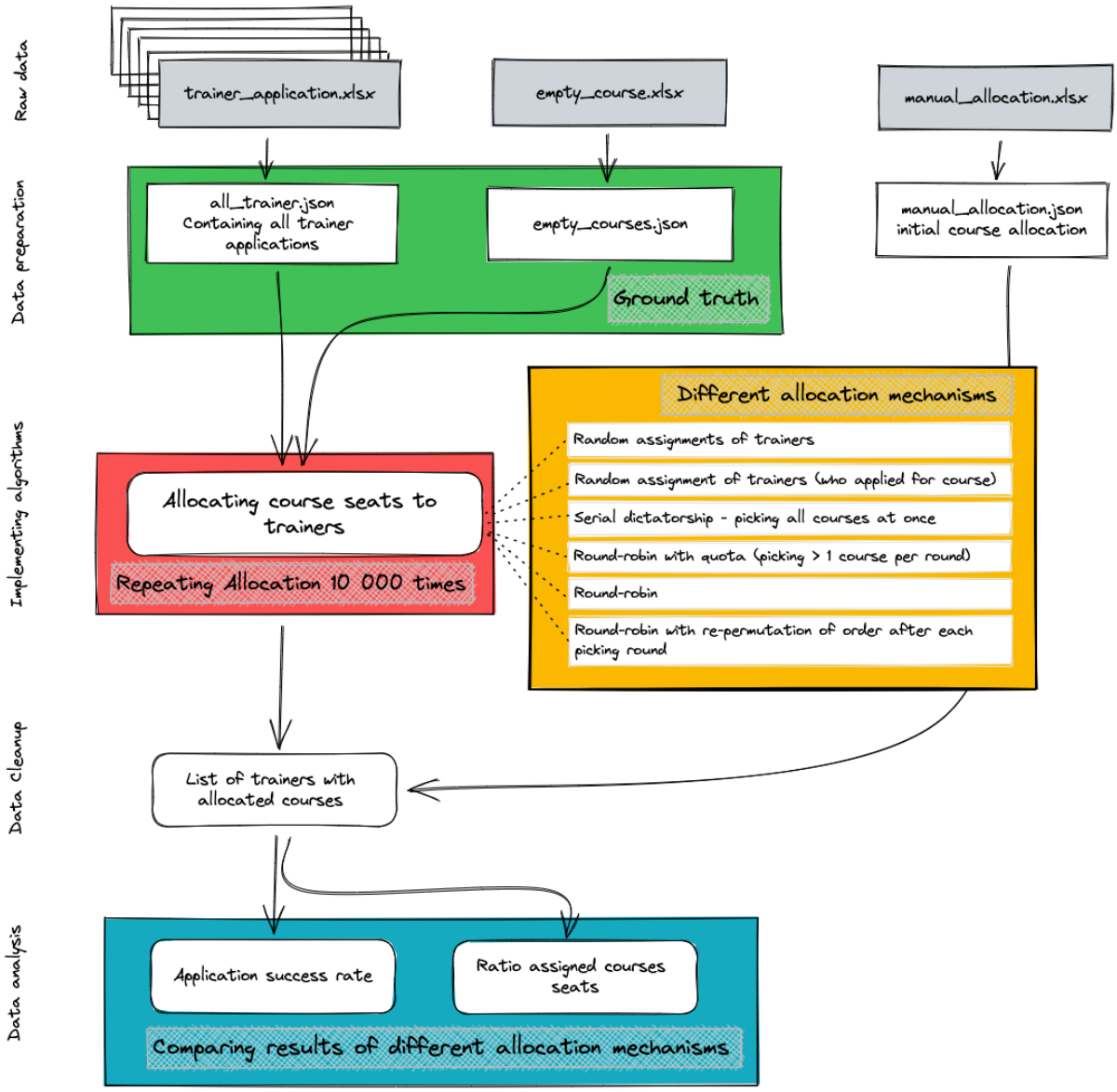
The first step, getting access to useful raw data, was harder than anticipated. I could not find any publicly available real-life data that contained an allocation problem of many-objects-to-one-agent. However, after a few requests, one organization that offers outdoor courses for students, voluntary workers, or trainees, was willing to provide me with data. Their data was stored in different Excel files; one spreadsheet contained all the courses, and each trainer has an extra Excel file with his_her preferences over the courses. In addition, I was provided with the 'manual' allocation of course seats to trainers. This allowed me not only to compare the results of different rule-based mechanisms among each other, but also with the manual allocation. Due to data protection reasons, sensitive data (i.e., names and pay rate) were pseudonymized. Due to the fact that trainers do not indicate a vector of preferences, but 'only' whether they are interested in a course or not, I randomized the courses into a preference order.

The second step was to process the raw data in a way that allowed me to implement the allocation mechanisms. In this step, the ground truth of this project was established. The Excel file with all unassigned courses was converted to a JSON file with the course ID as key and course information like starting date, end date, course type, pay rate as value. Each Excel file

of the trainers were parsed into one JSON file containing a list of trainers with all their course preferences. The preferences are stored as a list, thereby each element is a course and contains the relevant course information (e.g., course ID, preferences for courses). These two files were the input for each allocation mechanism, and thus the ground truth of this analysis project.

Figure 4

Workflow from raw data to data analysis (own illustration)



The different allocation mechanisms were implemented in one function. This function takes the two JSON files with defined structures and other boolean expressions that specify how the function allocates trainers and course seats. The individual steps of the allocation mechanisms are described in the previous chapter and are visualized in Figure 1-3. The output of the

function were two dictionaries that have the same structure as the input of the JSON files. The trainer dictionary contained information which of the individual preferred courses each trainer has allocated. The course dictionary contained information about which trainer has assigned to each course and which trainer has also applied for the respective course.

In the fourth step, the output dictionary of the allocations' function was converted to pandas data frames and summarized. This means for the trainer dictionary that each trainer got a count of how many courses he_she applied for and how many courses he_she allocated. Based on these two numbers per trainer, the individual application success rate was calculated (deviation). Further, the mean, standard deviation, and the coefficient of variation of the application success rate for all trainers as a whole was calculated. For the course dictionary, each course got a count of how many trainers were assigned to each course. These numbers are summed up to one value and divided by the sum of required trainers of each course. By diving those two metrics, the ratio of assigned course seats for the respective allocation was calculated. In an additional column of the data frames, all these key metrics were provided with information of which allocation mechanisms they were calculated. To counteract the random character of the allocation mechanism, the allocation was performed several times (in this project 5 000 times), which resulted in more robust results. The key metrics per allocation as well as the information about the respective allocation mechanisms were consolidated in a data frame, which is the basis for the data analysis. Furthermore, the key metrics were calculated for the manual allocation and appended to the data frame.

The last step of the project was to analyze the results. Based on the data frame (with mean, standard deviation, and the coefficient of variation of the application success rate and the ratio of assigned course seats of each performed allocation), a groupwise descriptive statistic was performed. Furthermore, independence tests of the individual key metrics were performed to test the hypothesis.

Description of Ground Truth / Project Data

The used data in this project included 48 courses, 25 trainers and 153 course days. The number of trainers that are required for each course differed between one to four. The total number of required trainer days were 390. In sum, 20 trainers applied for 542 course days. The four offered course types differ in their pay rate. The standard course is a multi-day school program (34 courses). Further, there are 5 one-day programs, 5 individual multi-day courses and 4 trainee courses. Table 1 summarizes the key figures.

Table 1

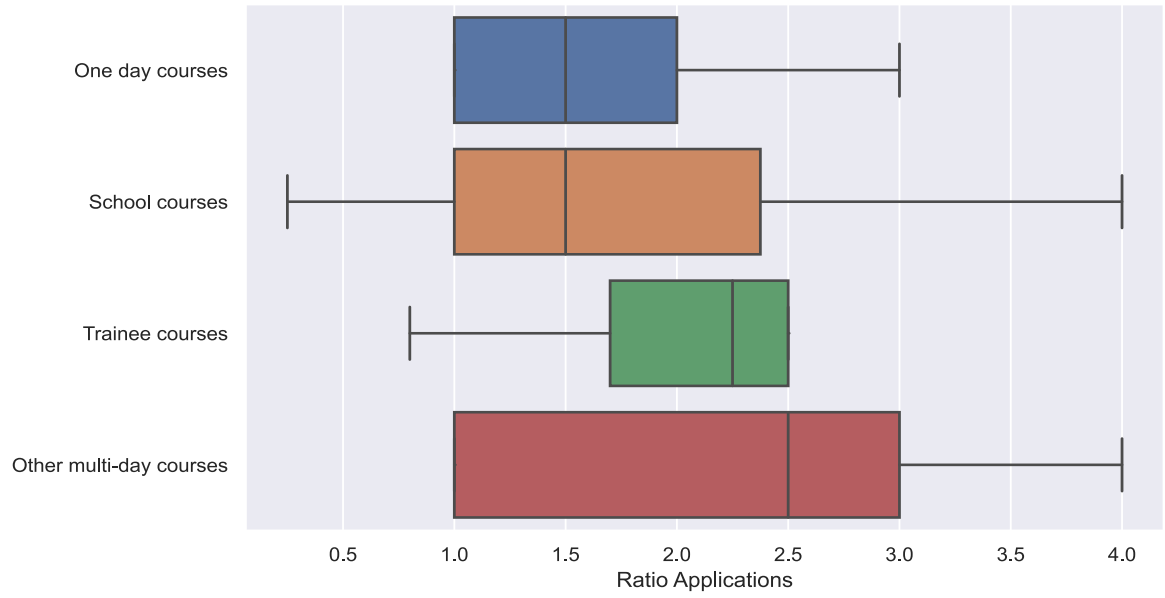
Key figures of the courses

Course Type	Pay Rate (in €) per Day	Number of Courses	Mean Duration (in Days)
School courses	288.8	34	3.2
One day courses	243.2	5	1.0
Individual courses	319.2	5	4.2
Trainee courses	395.2	4	3.8

For this analysis project, the most relevant information is how many applications were submitted per course. The boxplot below (Figure 5) illustrates the course type wise distributions of the ratio of applications ($= \frac{number\ of\ applicaitons}{number\ of\ required\ trainers}$). If we look at the distribution per course type, it shows that the course types 'one day courses' and 'individual courses' have always more applications than seats. A trainee course has less applications than seats and the lowest quantile of school courses have less applications than required trainers. In sum, nine course have fewer applications than required trainers.

Figure 5

Ratio of application per different course types (own illustration)



Regarding the trainer side of this allocation problem, 20 trainers (in total) applied for at least one course. The fast majority applied for 2-7 courses, and what is noteworthy is that two trainers applied for over 28 courses. Based on how many courses per course type each trainer

applied for, I performed a cluster analysis that yields three clusters (see Table 2). One cluster is formed by the trainers that applied for over 28 courses. The second cluster is characterized by those trainers that applied for the lowest number of courses in each category except for one day courses. The third cluster of trainers applied especially for long courses.

Table 2

Description of clusters

	Clusters		
	Few and short courses trainer	Long courses trainer	Workhorses
Number of Trainers	9	9	2
One day courses	0.44	0.22	2.50
Individual courses	0.11	1.56	1.50
School courses	4.56	3.67	25.00
Trainee courses	0.33	1.22	1.00
Duration	2.94	3.69	3.04
Total Number of Course Applications	5.44	6.67	30.00

Results, Discussion and Limitations

The aim of this project was to compare empirically the results of different allocation mechanisms that can be applied in many-goods-to-one-agent allocation problems. Therefore, I have implemented six different allocation mechanisms and applied them to real life data from an organization. The problem was the allocation of course seats (goods) to trainers (agents). Based on the specificity of the case, I have chosen two metrics which allowed an assessment of the efficiency of the allocation: (1) the ratio of assigned course seats and (2) the coefficient of variation, which combines the standard deviation as a measurement of fairness between trainers and the mean as a measurement of how many preferred courses seats were allocated per trainer. Each allocation was performed 10 000 times in order to get more observations. Table 3 summarizes the mean, minimum (best allocation) and standard deviation of coefficient of variation for each allocation type and the mean ratio of assigned course seats. It can be deduced that the most efficient allocation comes from the mechanism “Random from applications” (CV = 0.19). However, on average, the best results come from round-robin mechanisms with a low quota (CV = 0.32).

Table 3

Mean, minimum and standard deviation of CV and ratio assigned course seats per mechanism

Allocation Type	Coefficient of Variation			Ratio of Assigned Course Seats
	min	M	SD	M
Manual Allocation	0.35	0.35		0.81
Random from applications	0.19	0.36	0.05	0.90
Random from set of trainers	0.82	1.62	0.34	0.11
Round Robin	0.28	0.32	0.02	0.90
Round Robin Quota 2	0.25	0.32	0.03	0.90
Round Robin Quota 5	0.23	0.36	0.03	0.90
Round Robin re-permutated	0.28	0.32	0.02	0.90
Serial Dictatorship Picking all at once	0.30	0.54	0.08	0.90

The two boxplots below (Figure 6) show the distribution of all ratios of assigned course seats and CV. All allocation mechanisms except “Random from set of trainers” have the same ratio across all allocations. For the manual allocation, this can be explained by the fact that it was performed only once. For all other mechanisms, it can be explained by their logic. Those mechanisms are based on the assumption that each trainer wants to get as many as possible preferred courses, which means that the mechanism stops only if the courses are ‘full’ or no one is interested in available courses anymore. Hence, those mechanisms have the same ratio of assigned course seats. Regarding the CV, it is noticeable that the differences between the individual allocations are much smaller for the round-robin mechanisms than for SD and “Random from applications”. This impression is confirmed when looking at the scatterplot between mean and standard deviation of all allocations (Figure 7). It clearly shows that SD and “Random from applications” scatter much more than the round-robin mechanisms. Further, the lower the quota of allocated courses in each round, the lower the scatter.

Figure 6

Boxplots coefficient of variation, ratio of assigned course seats (own illustration)

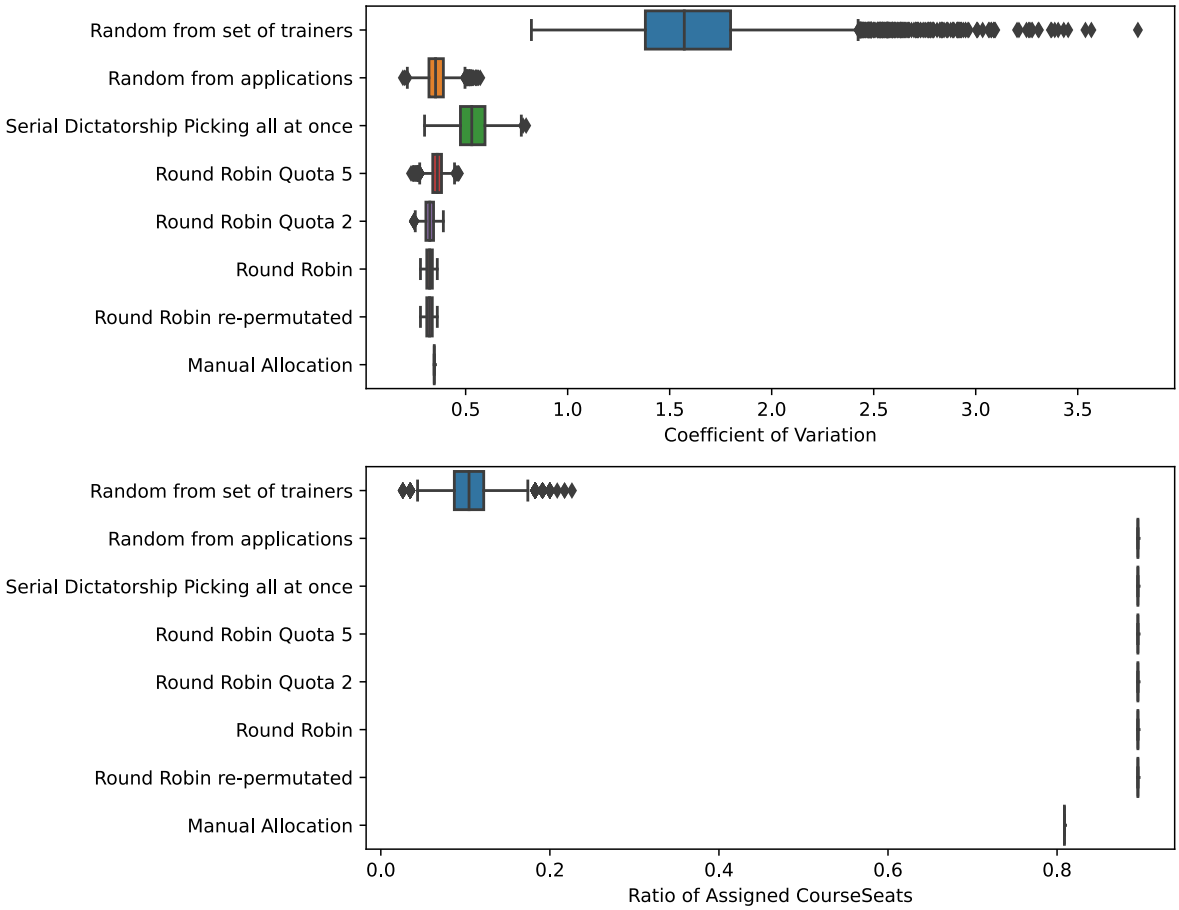
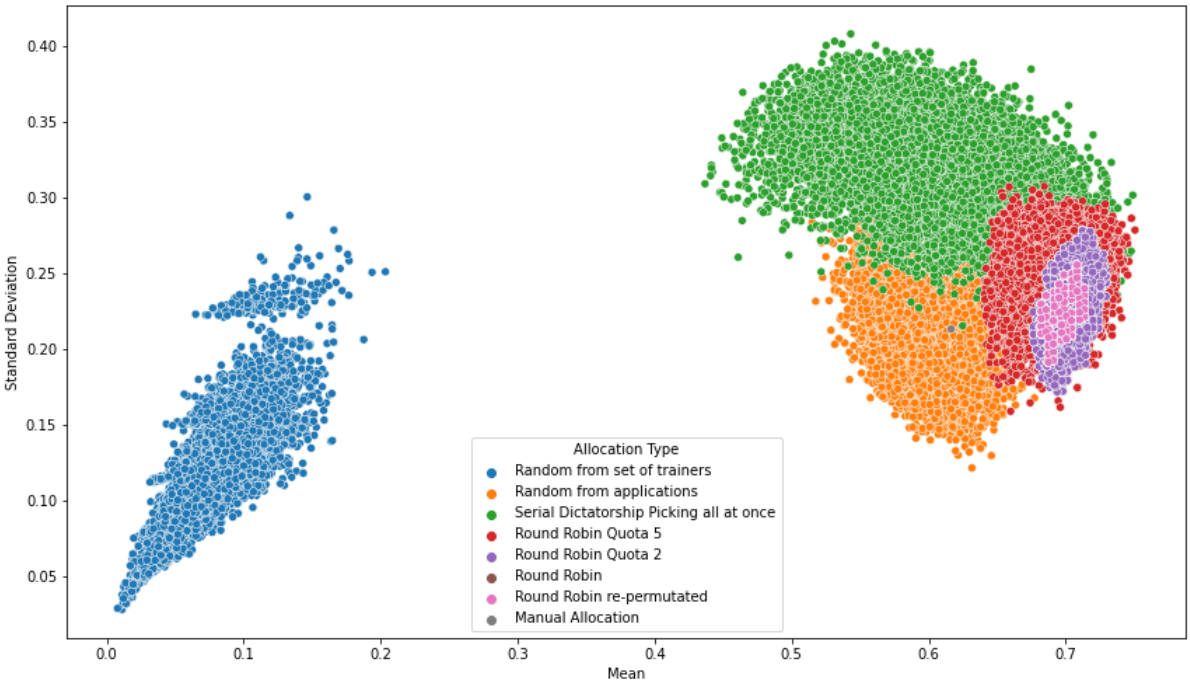


Figure 7

Scatterplot mean and standard deviation of CV (own illustration)



To test the set hypotheses, I performed both a mean based (one-way ANOVA) and a median based (Kruskal Wallis) independence test. In each case, I first tested location difference of all allocation mechanisms against each other for differences and removed the mechanism with the largest difference in the mean. Hence, the last model that I have tested compared the ratio of assigned course seats and the CV of application success rate between round-robin with re-permutation and round-robin without re-permutation. Regarding the results of the independency test for the ratio of assigned course seats, Table 4 shows that there is a significant difference between "All mechanisms" and further between all mechanisms except the "Random from set of trainers". As described above, due to the logic of the mechanisms, all mechanisms (except "Random from set of trainers" and the manual allocation) yield the same ratio of assigned course seats. Thus, the Kruskal Wallis test and the one-way ANOVA test cannot be performed because all allocation mechanisms have literally the same values. Following, H_0 1 can partly be refuted in the sense that the random allocation from all trainers without considering their preferences and the manual allocation yield significant different ratios of assigned course seats to all other mechanisms.

Table 4

Independency tests of ratio of assigned course seats between the mechanisms

Allocation Type	Kruskal Wallis		One-way ANOVA	
	H-statistics	p-value	F-statistics	p-value
All Mechanisms	69453.8083	>0.001	6982738.2610	>0.001
Excluding Random from set of trainers	60000.0000	>0.001	inf	>0.001

Table 5 summarizes the results of the independency tests for the CV of the application success rate. Based on the p-value of both the Kruskal Wallis test and the one-way ANOVA test, the location between all allocation mechanisms differs. However, it should be mentioned that if only 100 allocations per mechanisms are performed, the independency tests of all round-robin mechanisms are no longer significant.

Nevertheless, based on the performed independency tests, H_0 2 can be refuted. However, the CV of the application success rate of the round-robin mechanisms did not differ significantly from each other up to a certain number of retrievals. Remembering the small nominal differences of the CV of the round-robin mechanisms with a quota of at most two, I would argue that it is not so much the empirical differences of the mechanisms that are decisive which one is applied in practice but rather whether it is important to the decision-makers to reduce the

random character by a picking quota of one good (envyfreeness up to one good) or the re-permutation after each picking round or not.

Table 5

Independency tests of CV of application success rate between the mechanisms

Allocation Type	Kruskal Wallis		One-way ANOVA	
	H-statistics	p-value	F-statistics	p-value
All Mechanisms	49178.1191	>0.001	108763.6883	>0.001
Excluding Random from set of trainers	31936.8834	>0.001	28459.1548	>0.001
+ excluding Manual Allocation	31936.2755	>0.001	34150.9278	>0.001
+ excluding Random from applications	30020.4881	>0.001	45878.9730	>0.001
+ excluding Serial Dictatorship Picking all at...	9913.5028	>0.001	5619.1008	>0.001
+ excluding Round Robin Quota 5	97.9356	>0.001	55.3285	>0.001
+ excluding Round Robin Quota 2	98.5569	>0.001	100.2669	>0.001

Last but not least, this analysis has some limitations. The presented analysis refers to only one preference vector per trainer. It would have been very interesting to randomly simulate other preference vectors and see how the results of the mechanisms change. However, this was not feasible due to the scope of this project. Furthermore, it is questionable whether the CV is a suitable measure for the efficiency of a mechanism. It would be better if the agents would not only give an ordinary ranking of the courses but also a utility per course. This would make it much easier to compare how the benefit changes for the individual trainers or for all of them together. However, questions such as how many “utility points” each agent can assign, or do they all have the same amount, etc. would then have to be clarified. In addition, the mechanism would become much more complex, and thus more difficult to implement in practice.

In sum, the aim of the project was to explore a many-goods-to-one-agent allocation problem empirically. Six different allocation mechanisms have been implemented. For each allocation, a coefficient of variation (= a measurement of the spread of values (fairness) relative to the mean (performance)) has been calculated and was used as measurement of the efficiency of each allocation. Each mechanism was performed 10 000 times in order to get more observations. Comparing the location difference of the coefficients of variation between the different types of mechanisms, the independency tests yield significance difference between all mechanisms. However, the nominal difference between the round-robin allocations with a quota of maximal two is negligible which is why a mechanism can be chosen depending on

the preferences of the decision-makers. Due to the logic of the implemented mechanisms, the difference of ratio of assigned course seats does only differ between the “Random from set of trainers” and if the allocation is done manually (without following an algorithm). All other mechanisms yield the exact same ratio of assigned course seats.

In cases that have the same characteristics as the analyzed data, the following practical implication can be given: As soon as one uses a mechanism that algorithmically incorporates the preferences of the trainers, then the ratio of the allocated courses is always the same. This implies that the choice of algorithm can only improve the allocation from the trainers’ point of view. However, if you do not have the possibility to do different allocations multiple times (in this project 10 000 times), then round-robin with a low quota (below 3) is a very efficient mechanism for trainers to allocate as many preferred course seats as possible. However, if there is an opportunity to try different allocations multiple times and select the best one, then it is worth integrating other mechanisms as well. Automatic allocations make it possible to try as many allocations as you like, allowing for more efficient allocations (compared to manual allocation!). In order to better assess the utility of an allocation, trainers should also submit an assumed utility to their preferred courses. This may well change which mechanism is the most efficient.

Appendix

Most relevant files

Step in Workflow	Moduls that cover the step	script that perform the step
Data preparation / formatting	M1, M2	rawdata_to_GT.py
Analyzing ground truth	M3	analysis_GT.ipynb
Implementing allocation mechanisms	M1, M4	implementing_mechanisms_data_cleaning.py
Data Cleaning for Analysis	M1, M2 (, M3)	implementing_mechanisms_data_cleaning.py
Data Analysis	M3, M4	data_analysis.ipynb
All relevant functions	M1, M2, M3	data_parsing_functions.py

rawdata_to_GT.py

```
# This Script establish the ground truth for this project analysis,
# it process the raw excel files from trainers and empty courses into two
json files
# (all_courses_empty.json // trainer_applications.json)

# Importing Moduls:
import glob
import regex
import json
from data_parsing_functions import *

if __name__ == '__main__':
    # ------------------------------------------------------------------------
---------------
    # Check if all required directory exists:
    check_create_dir(DATA_PATH)
    path_trainer_applications = check_create_dir(DATA_PATH,
DIR_TRAINER_WISHES)

    # ------------------------------------------------------------------------
---------------
    # Parsing Raw Excel files to JSON - GROUND TRUTH:
    parsing_trainer_applications = input("Do you want to parse trainers
application and empty courses and store them as json file? \nif YES enter
Y/y \nif not enter any other key: ")
    if parsing_trainer_applications.lower() in ["y", "yes"]:
        ALL_trainers_applications = []
        trainer_files = glob.glob(path_trainer_applications+"/*")

        for i in trainer_files:
            trainer_name_file =
regex.sub(rf"^{path_trainer_applications}/","",i)
            trainer_name = regex.sub(r".xlsx","",trainer_name_file)
            trainer_dic =
trainer_app_excel_to_dict(path_trainer_applications, trainer_name)
            ALL_trainers_applications.append(trainer_dic)

        with open(f"{DATA_PATH}/trainer_applications.json", "w") as
outfile:
```

```
            json.dump(ALL_trainers_applications, outfile)
        print(f"All Trainer Applications are parsed and stored in
{DATA_PATH}/trainer_applications.json")

        empty_courses = trainer_app_excel_to_dict("./data")
        with open(f"{DATA_PATH}/all_courses_empty.json", "w") as outfile:
            json.dump(empty_courses, outfile)
            print(f"All empty courses are parsed and stored in
{DATA_PATH}/all_courses_empty.json")
    else:
        print("NO raw excel files are parsed")
```

implementing_mechanisms_data_cleaning.py

```
# This Script allocates trainers and course seats based on the GT created
by the script rawdata_to_GT.py

# Importing Moduls:
import json
from data_parsing_functions import *


if __name__ == '__main__':
    # ------------------------------------------------------------------------
---------------
    # Check if all required directory exists:
    check_create_dir(DATA_PATH)
    path_trainer_applications = check_create_dir(DATA_PATH,
DIR_TRAINER_WISHES)
    path_data_analysis = check_create_dir(DATA_PATH,
DIR_DATA_CLEANED_for_ANALYSIS)
    path_results = check_create_dir(RESULTS_PATH)

    #### DICT of all Trainer Applications:
    with open(f"{DATA_PATH}/trainer_applications.json", "r") as in_file:
        all_trainers = json.load(in_file)

    #### DICT of all courses:
    with open(f"{DATA_PATH}/all_courses_empty.json", "r") as in_file:
        all_empty_courses = json.load(in_file)


    #### DICT of MANUAL course Allocation:
    with open(f"{DATA_PATH}/manual_allocation.json", "r") as in_file:
        manu_allocation = json.load(in_file)

    ######### Applying all implemented allocation mechanisms with re-
allocating several times
    rounds= 1000
    best_allocation_threshold = 999
    overall_best_allocation = None

    lst_all_key_metrics = []
    list_all_sucess_rates = []
    ## Name , how many times repeating allocation , re permutate
true/false, quota per pick, random application from interested trainers,
random application from set of trainers
    for all_type in [("Random from set of trainers", rounds, False, 1,
False, True),
                     ("Random from applications", rounds, False, 1, True,
False),
```

```
                    ("Serial Dictatorship", rounds,False, "ALL" ,False,
False),
                    ("Round Robin", rounds, False, 5 ,False, False),
                    ("Round Robin", rounds, False, 2 ,False, False),
                    ("Round Robin", rounds, False, 1 ,False, False),
                    ("Round Robin", rounds, True, 1 ,False, False),

                    ]:
        ## Performing and cleaning all allocation mechanisms
        df_key_metrics, df_success_rate, best_allocation =
repeat_allocation_return_df(all_trainers, all_empty_courses, all_type[0],
all_type[1], re_permutate=all_type[2],
quota_per_pick=all_type[3],random_from_applications=all_type[4],
random_total=all_type[5])
        lst_all_key_metrics.append(df_key_metrics)
        list_all_sucess_rates.append(df_success_rate)
        print("The best allocation for the mechanisms ",best_allocation[0],
f"yields to {coef_variation} of: ", best_allocation[1])
        if best_allocation[1] < best_allocation_threshold:
            overall_best_allocation = best_allocation
            best_allocation_threshold = best_allocation[1]

    ## Parsing manual Allocation to other allocations:
    manual_application_succcess_rate, manual_stats =
dicts_to_application_success_rate_AND_key_metrics(all_trainers,
manu_allocation)
    manual_application_succcess_rate[allocation_type]  = "Manual
Allocation"
    manual_stats = [manual_stats]
    df_manual_stats = pd.DataFrame(manual_stats, columns=[coef_variation,
mean, stand_dev, name_ratio_filled_seats])
    df_manual_stats[allocation_type] = "Manual Allocation"

    lst_all_key_metrics.append(df_manual_stats)
    list_all_sucess_rates.append(manual_application_succcess_rate)

    # ------------------------------------------------------------------
---------------
    # Combining clean dataframes of all allocation mechanisms
    all_key_metrics = pd.concat(lst_all_key_metrics, ignore_index = True)
    all_sucess_rates = pd.concat(list_all_sucess_rates, axis=0,
ignore_index = True)

    # ------------------------------------------------------------------
---------------
    # Saving cleaned dataframes to csv files
    all_key_metrics.to_csv(f"{path_data_analysis}/{name_key_metrics}",
index = False)

all_sucess_rates.to_csv(f"{path_data_analysis}/{name_all_success_rates}",
index = False)

    # ------------------------------------------------------------------
---------------
    # Storing best allocation as
    with
open(f"{path_results}/best_allocation_with_{overall_best_allocation[0]}.jso
n", "w") as outfile:
            json.dump(overall_best_allocation[3], outfile)
            print(f"All empty courses are parsed and stored in
{DATA_PATH}/all_courses_empty.json")
```

```
overall_best_allocation[2].to_csv(f"{path_results}/best_allocation_{app_suc
cess_rate}.csv", index = False)

    print("\nDONE!")
```

data_parsing_functions.py

```
from aiohttp import BadContentDispositionParam
import pandas as pd
import numpy as np
import random
import copy
import os


### Name Definitions!!
DATA_PATH = "data"
DIR_TRAINER_WISHES = "trainer_files"
DIR_DATA_CLEANED_for_ANALYSIS = "data_for_analysis"
name_key_metrics = "all_key_metrics_from_all_allocations.csv"
name_all_success_rates = "all_success_rates_from_all_allocations.csv"
RESULTS_PATH = "results"
RESULTS_PATH_abs = "/Users/ritchyk/Library/Mobile
Documents/iCloud~md~obsidian/Documents/ObsiNotes/_Attachments"
id = "ID"
start = "Start"
end = "End"
duration = "Duration"
prio_application = "Prio Application"
course_class = "Course_type"
client = "Client"
pay_rate = "Pay Rate"
number_trainers_required = "Number Trainers Required"
number_asigned_trainers = "Number Assigned Trainers"
app_success_rate = "Application Success Rate"
application = "Application"
assigned = "Assigned"
wait_list = "Wait list"
app_success_rate = "Application Success Rate"
mean = "Mean"
stand_dev = "Standard Deviation"
coef_variation = "Coefficient of Variation"
name_ratio_filled_seats = "Ratio of Assigned CourseSeats"
allocation_type = "Allocation Type"


## Function for checking and creating directory:
def check_create_dir (*a_path):
    """Thus function checks if a specific directory existis and if not it
is created

    Returns:
        str: a directory in the working directory
    """
    try:
        path_exist = os.path.join("./", a_path[0], a_path[1])
    except:
        path_exist = os.path.join("./", a_path[0])
    data_analysis_dir_exists = os.path.exists(path_exist)
    if data_analysis_dir_exists:
        print(f"\nThe path {path_exist} exists")
```

```
    else:
        os.mkdir(path_exist)
        print(f"\nThe path {path_exist} did not exist and was created")
    return path_exist

# Data Processing
def trainer_app_excel_to_dict (a_path, a_name = None):
    """This function convertes all trainer excel or if a_name == None the
empty course files that are structured in a specific and equal way into a
json file

    Args:
        a_path (str): The path of the directory of all trainer files
        a_name (str, optional): the name of the file without the ending.

    Returns:
        dic: A dictionary containing all trainers and their preferences.
    """
    column_names = [id, start, end, course_class, client, pay_rate,
number_trainers_required]
    if a_name == None:
        try:
            excel_df = pd.read_excel(f"{a_path}/All_courses_empty.xlsx")
        except:
            print(f"{a_path}/All_courses_empty.xlsx Does not exist pls
check in the {a_path} directory")
        excel_df[assigned] = np.empty((len(excel_df),0)).tolist()
        excel_df[wait_list] = np.empty((len(excel_df),0)).tolist()
        excel_df[number_asigned_trainers] = 0
        column_names.append(number_asigned_trainers)
        column_names.append(assigned)
        column_names.append(wait_list)

    else:
        excel_df = pd.read_excel(f"{a_path}/{a_name}.xlsx")
        excel_df = excel_df.rename(columns= {"id": id, "start": start,
"end": end, "course_type": course_class, "client": client, "pay_rate":
pay_rate, "number_trainers_required": number_trainers_required,
"application": application})

        column_names.append(application)
    excel_df = excel_df.rename(columns= {"Hotelbuchungsnummer": id,
"Beginn": start, "Ende": end, "Bildungsprogramm": course_class, "Gruppe":
client, "Tagessatz": pay_rate, "Ben√∂tigte Trainer":
number_trainers_required, "Wunsch": application})
    updated_df = excel_df.copy()


    #Renaming the columns
    updated_df = updated_df[column_names]
    updated_df = updated_df.dropna(subset=[pay_rate])
    # Only parsing rows where trainer is interested in the course
    if a_name != None:
        updated_df = updated_df.dropna(subset=[application, pay_rate])
        # only neede if no prio_application submitted
        updated_df = updated_df.sample(frac=1)
        updated_df.reset_index(inplace=True, drop=True)
        updated_df[prio_application] = updated_df.index+1
    updated_df[duration] = (pd.to_datetime(updated_df[end]) -
pd.to_datetime(updated_df[start])).dt.days +1
    try:
        updated_df[end] = updated_df[end].dt.strftime('%Y-%m-%d')
        updated_df[start] = updated_df[start].dt.strftime('%Y-%m-%d')
```

```
    except:
        pass



    if a_name == None:
        updated_df.set_index(id, inplace=True, drop=True)
        trainer_dic = updated_df.to_dict(orient= "index")
        return trainer_dic
    else:
        trainer_dic = updated_df.to_dict(orient= "records")
        return {"name": a_name, "pick_order": [], "courses": trainer_dic}

############################################################
### Function -> applying different allocation mechanisms  ####
############################################################

def applying_allocation_mechanism(trainer_dict, all_courses_dict, a_name,
re_permutate = False, quota_per_pick = 1, random_from_applications = False,
random_total = False):
    """This function implements different allocation mechanisms and returns
two dictionaries one containg the allocation from the course side (dic of
assigned trainers) and from trainers side - (dic of trainers and allocated
courses)

    Args:
        trainer_dict (dic): a dictionary of all trainers applications (from
function:trainer_app_excel_to_dict )
        all_courses_dict (dic): a dictionary of all empty courses (from
function:trainer_app_excel_to_dict )
        a_name (str): a name that is moved through the function
        re_permutate (bool, optional): Specifing if the order of trainers
is repermutated after each picking round. Defaults to False.
        quota_per_pick (int, optional): Number of courses that can be
picked in each round. Defaults to 1.
        random_from_applications (bool, optional): If True -> allocationg
from " course side" picking as much trainers as required from all trainers
that applied for this course . Defaults to False.
        random_total (bool, optional): If True -> allocationg from " course
side" picking as much trainers as required from all trainers in trainers
dic. Defaults to False.

    Returns:
        dic: two dictionaries (dic of assigned trainers) (dic of trainers
and allocated courses)
    """
    a_trainer_dict = copy.deepcopy(trainer_dict)
    a_all_courses_dict = copy.deepcopy(all_courses_dict)

    counter = None
    random.shuffle(a_trainer_dict) #random permutation of the picking order
    while counter != 0:
        counter = 0
        for pick_order, trainer in enumerate(a_trainer_dict):
            trainer["pick_order"].append(pick_order+1)

            if random_from_applications == True or random_total == True:  #
Allocation from course side perspective
                for wanted_course in trainer["courses"]:
                    course = wanted_course[id]
                    if assigned not in wanted_course.keys():
                        wanted_course[assigned] = "No"
                    ## Assigning all to waitlist
```

```
                    if wanted_course[assigned] == "No":
                        counter += 1

a_all_courses_dict[course][wait_list].append(trainer["name"])
                        wanted_course[assigned] = wait_list

            elif random_from_applications == False and random_total ==
False:
                lowest_prio = 999
                prefered_course = None

                # Serial Dictatorship: all preferences of a trainer are
assigned at once
                if quota_per_pick == "ALL":
                    for wanted_course in trainer["courses"]:
                        course = wanted_course[id]
                        if assigned not in wanted_course.keys():
                            wanted_course[assigned] = "No"
                        ## Conditions for assining the most prefered course
                        if wanted_course[assigned] == "No":
                            if
a_all_courses_dict[course][number_trainers_required] >
a_all_courses_dict[course][number_asigned_trainers]:

a_all_courses_dict[course][assigned].append(trainer["name"])
                                wanted_course[assigned] = "Yes"

a_all_courses_dict[course][number_asigned_trainers] += 1
                            else:

a_all_courses_dict[course][wait_list].append(trainer["name"])
                                wanted_course[assigned] = wait_list
                else:
                    ## Round Robin implementation
                    for i in range(quota_per_pick): # With different
quotas, if 1 -> classical Round Robin
                        for wanted_course in trainer["courses"]:
                            if assigned not in wanted_course.keys():
                                wanted_course[assigned] = "No"
                            ## Conditions for assining the most prefered
course
                            if wanted_course[assigned] == "No":
                                counter +=1
                            if wanted_course[prio_application] <=
lowest_prio and wanted_course[assigned] == "No":
                                lowest_prio =
wanted_course[prio_application]
                                prefered_course = wanted_course
                            if lowest_prio == 999:
                                continue
                        if prefered_course != None:# and
a_all_courses_dict[course]:
                            ## Assigning the trainer to the prefered course
or Waitlist
                            course = prefered_course[id]
                            if
a_all_courses_dict[course][number_trainers_required] >
a_all_courses_dict[course][number_asigned_trainers]:

a_all_courses_dict[course][assigned].append(trainer["name"])
                                prefered_course[assigned] = "Yes"

a_all_courses_dict[course][number_asigned_trainers] += 1
```

```
                            else:

a_all_courses_dict[course][wait_list].append(trainer["name"])
                                prefered_course[assigned] = wait_list
                        prefered_course = None
                        lowest_prio = 999

        if re_permutate == True:
            random.shuffle(a_trainer_dict) #re-permutate the picking order

    ## Allocation from course side perspective
    # Random from trainers who applied for the course
    if random_from_applications == True and random_total == False:
        for each_course in a_all_courses_dict:
            course = a_all_courses_dict[each_course]
            interest = course[wait_list]
            random.shuffle(interest)
            needed_trainer = int(course[number_trainers_required])
            for i in range(needed_trainer):
                try:
                    course[assigned].append(interest[i])
                    course[number_asigned_trainers] += 1
                except:
                    pass

    # Total Random from all possible trainers
    elif random_total == True and random_from_applications == False:
        for each_course in a_all_courses_dict:
            course = a_all_courses_dict[each_course]
            pot_interest = [x["name"] for x in a_trainer_dict]
            random.shuffle(pot_interest)
            needed_trainer = int(course[number_trainers_required])
            for i in range(needed_trainer):
                if pot_interest[i] in course[wait_list]:
                    course[assigned].append(pot_interest[i])
                    course[number_asigned_trainers] += 1
                else:
                    pass
    elif random_total == True and random_from_applications == True:
        print("This specification is not possilble check the arguments of
random_total and random_from_applications must no be both TRUE")

    ## Returning overall picking order the smaller the number the earlier
you were involved in the picking order
    for trainer in a_trainer_dict:
        trainer["pick_order"] =
sum(trainer["pick_order"])/len(trainer["pick_order"])
    return a_trainer_dict, a_all_courses_dict



## Converting Trainersapplications -> to dataframe
def dict_trainer_to_df(a_dict):
    """This function converts a trainers dic to an pandas dataframe

    Args:
        a_dict (dic): dic from function: applying_allocation_mechanism

    Returns:
        dataframe: A dataframe of trainers applications and assignments
    """
    try:
```

```
        trainer_df = pd.json_normalize(a_dict, "courses", ["name",
"pick_order"])
    except:
        trainer_df = pd.json_normalize(a_dict, "courses", ["name"])
    return trainer_df

def dict_allocation_to_df (a_dict):
    """Thus function converts a course dic to an pandas dataframe

    Args:
        a_dict (dic): dic from function: applying_allocation_mechanism

    Returns:
        dataframe: A dataframe of courses wheres assigned column is a list
    """
    df = pd.DataFrame.from_dict(a_dict, orient="index")
    return df

def df_with_lst_to_sep_rows (a_df, a_column):
    """Thus function extents an column with list as values into separate
rows

    Args:
        a_df (dataframe): a dataframe from the function
dict_allocation_to_df
        a_column (str): name of the column

    Returns:
        dataframe: a dataframe with separate rows for each element of a
list of a column
    """
    exploded_df = a_df.copy()
    exploded_df = exploded_df.explode(a_column)
    return exploded_df

def df_application_success_per_trainer (a_trainer_df, a_allocation_df):
    """this functions calculates the application success rate for each
trainer of an allocation

    Args:
        a_trainer_df (dataframe): a dataframe from the function:
dict_trainer_to_df
        a_allocation_df (dataframe ):a dataframe from the function:
df_with_lst_to_sep_rows

    Returns:
        dataframe: containing the application success rate for each trainer
of the allocation
    """
    try:
        trainer_sum = a_trainer_df.groupby("name").agg({"pick_order":
"mean", pay_rate: "mean", duration: "sum", application: "count"})
    except:
        trainer_sum = a_trainer_df.groupby("name").agg({pay_rate: "mean",
duration: "sum", application: "count"})

    trainer_sum = trainer_sum.add_prefix("applied_")
    trainer_group_assigned =
a_allocation_df.groupby(assigned).agg({pay_rate: "mean", duration: "sum",
assigned: "count"})
    combined_df = pd.concat([trainer_sum, trainer_group_assigned], axis=1)
    combined_df = combined_df.fillna(0)
```

```
    combined_df[app_success_rate] =
combined_df[assigned]/combined_df[f"applied_{application}"]
    return combined_df

def dicts_to_application_success_rate_AND_key_metrics (a_trainer_dict,
a_allocation_dict):
    """combining the different functions from trainers, allocation dict to
dataframes in one step and calculates the key metrics for the research
project

    Args:
        a_trainer_df (dataframe): a dataframe from the function:
dict_trainer_to_df
        a_allocation_df (dataframe ):a dataframe from the function:
df_with_lst_to_sep_rows

    Returns:
        tuple: dataframe with application success rate for each trainer and
tuple with coefficient of variation, mean, standard deviation and ratio of
assigned course seats of the respective allocation)
    """
    trainer_df = dict_trainer_to_df(a_trainer_dict)
    alloc_df = dict_allocation_to_df(a_allocation_dict)
    if number_asigned_trainers not in alloc_df.columns:
        alloc_df[number_asigned_trainers] = alloc_df[assigned].str.len()
    sum_required_trainers = alloc_df[number_trainers_required].sum()
    sum_filled_course_seats = alloc_df[number_asigned_trainers].sum()
    ratio_filled_seats = sum_filled_course_seats / sum_required_trainers
    alloc_explod = df_with_lst_to_sep_rows(alloc_df, assigned)
    trainer_success_rate_df =
df_application_success_per_trainer(trainer_df, alloc_explod)
    mean, std = (trainer_success_rate_df[app_success_rate].mean(),
trainer_success_rate_df[app_success_rate].std())
    coef_of_varia = std/mean
    return trainer_success_rate_df, (coef_of_varia, mean, std,
ratio_filled_seats)


def repeat_allocation_return_df (trainer_dict, empty_course_dict,
name_of_mechanism, num_allocations = 100 ,re_permutate=False,
quota_per_pick = 1, random_from_applications = False, random_total =
False):
    """This function combines all the implementing and cleaning functions
from abaove in one step, furthermore it integrates the possiblity the
repeat the allocation as often as wanted

    Args:
        trainer_dict (dic): a dictionary of all trainers applications (from
function:trainer_app_excel_to_dict )
        all_courses_dict (dic): a dictionary of all empty courses (from
function:trainer_app_excel_to_dict )
        name_of_mechanism (str): a name that is moved through the function
        num_allocations (int): the number of times the allocation is to be
repeated
        re_permutate (bool, optional): Specifing if the order of trainers
is repermutated after each picking round. Defaults to False.
        quota_per_pick (int, optional): Number of courses that can be
picked in each round. Defaults to 1.
        random_from_applications (bool, optional): If True -> allocationg
from " course side" picking as much trainers as required from all trainers
that applied for this course . Defaults to False.
```

```
        random_total (bool, optional): If True -> allocationg from " course
side" picking as much trainers as required from all trainers in trainers
dic. Defaults to False.

    Returns:
        a dataframe of all keymetrics of all allocation of this type of
allocation
        // a dataframe of all application success rates for all allocations
        // the most efficiant allocaiton of this time (measured by the
coefficient of variation)   (name of allocation, coef_variation, dataframe
of application success rate, allocation as dict)
    """
    best_allocation_threshold = 999
    best_allocation = None

    finding_optimum_lst = []
    df_sucess_rate_lst = []
    for i in range(num_allocations):
        trainers, course_all = applying_allocation_mechanism(trainer_dict,
empty_course_dict, name_of_mechanism, re_permutate=re_permutate,
quota_per_pick = quota_per_pick, random_from_applications =
random_from_applications, random_total = random_total)
        tmp_df , tmp_key_metrics =
dicts_to_application_success_rate_AND_key_metrics(trainers, course_all)
        tmp_df.reset_index(inplace=True)
        if tmp_key_metrics[0] < best_allocation_threshold:
            best_allocation = [tmp_key_metrics[0], tmp_df, course_all]
            best_allocation_threshold = tmp_key_metrics[0]
        finding_optimum_lst.append(tmp_key_metrics)
        df_sucess_rate_lst.append(tmp_df)
    df_optimum = pd.DataFrame(finding_optimum_lst, columns=[coef_variation,
mean, stand_dev, name_ratio_filled_seats])

    df_all_app_success_rates =  pd.concat(df_sucess_rate_lst, ignore_index
= True)
    if quota_per_pick == 1:
        quota = ""
    elif quota_per_pick == "ALL":
        quota = " Picking all at once"
    else:
        quota = f" Quota {str(quota_per_pick)}"
    if re_permutate == False:
        re_perm = ""
    elif re_permutate == True and quota_per_pick == "ALL":
        re_perm = ""
    elif re_permutate == True:
        re_perm = " re-permutated"
    df_all_app_success_rates[allocation_type] =
f"{name_of_mechanism}{re_perm}{quota}"
    df_optimum[allocation_type] = f"{name_of_mechanism}{re_perm}{quota}"
    best_allocation.insert(0, f"{name_of_mechanism}{re_perm}{quota}")
    return df_optimum, df_all_app_success_rates, best_allocation

def stand_df(a_df, a_name_colum=None):
    """Thus function standardizes an dataframe (mean = 0) if value > +/-1 -
-> value more than 1std from mean away

    Args:
        a_df (dataframe): a dataframe
        a_name_colum (str, optional): Name of column to standardize .
Defaults to None.

    Returns:
```

```
        dataframe: df with standardized values
    """

    vnames = [name for name in globals() if globals()[name] is a_df]
    #Normalizing data
    print(f"Standardizing the dataframe {vnames}")
    stand_df = a_df.copy()
    for name in a_df:
        if name == a_name_colum:
            continue
        stand_df[name] = (a_df[name] -
stand_df[name].mean())/stand_df[name].std()
    return stand_df
```

References

Abdulkadroğlu, A., Angrist, J. D., Narita, Y., Pathak, P. A., & Zarate, R. A. (2017). Regression Discontinuity in Serial Dictatorship: Achievement Effects at Chicago's Exam Schools. *American Economic Review*, *107*(5), 240–245. https://doi.org/10.1257/aer.p20171111

Biswas, A., & Barman, S. (2019). Matroid Constrained Fair Allocation Problem. *Proceedings of the AAAI Conference on Artificial Intelligence*, *33*(01), 9921–9922. https://doi.org/10.1609/aaai.v33i01.33019921

Caragiannis, I., Kurokawa, D., Moulin, H., Procaccia, A. D., Shah, N., & Wang, J. (2016). The Unreasonable Fairness of Maximum Nash Welfare. *Proceedings of the 2016 ACM Conference on Economics and Computation*, 305–322. https://doi.org/10.1145/2940716.2940726

Lien, J. W., Zheng, J., & Zhong, X. (2017). Ex-ante fairness in the Boston and serial dictatorship mechanisms under pre-exam and post-exam preference submission. *Games and Economic Behavior*, *101*, 98–120. https://doi.org/10.1016/j.geb.2016.07.003

Manea, M. (2007). Serial dictatorship and Pareto optimality. *Games and Economic Behavior*, *61*(2), 316–330. https://doi.org/10.1016/j.geb.2007.01.003

Svensson, L.-G. (1999). Strategy-proof allocation of indivisible goods. *Social Choice and Welfare*, *16*(4), 557–567.