Operations Research, Spring 2024 (112-2)
Midterm Project

Instructor: Ling-Chieh Kung
Department of Information Management
National Taiwan University

1 Submission rules

- This midterm project is due at ***23:59, May 4***. Submissions late by no more than twelve hours get 10 points off; those late by no more than 24 hours get 20 points off; those late by more than 24 hours get no point.
- For this midterm project, students should work in teams. Unless a student gets special approval from the instructor, her/his team members should be the same as those for the final project.
- For each team, ***one and exactly one*** student should submit their work on behalf of all team members. Please submit a ***ZIP file*** containing a ***PDF file*** and a ***Python program*** (i.e., a .py file) through NTU COOL. Make sure that the submitted PDF file contains the student IDs and names. Those who fail to do these will get 10 points off.
- The Python program should follow the ***Python 3.9*** standard and all requirements specified in this project. If a team's Python program does not satisfy the requirements so that it cannot be directly correctly by the TAs, the team may request for a resubmission in 48 hours after receiving notices from the TAs. However, the resubmission will make the team lose 10 points.
- You are required to ***type*** your work with LaTeX (strongly suggested) or a text processor with a formula editor. Hand-written works are not accepted. You are responsible to make your work professional in mathematical writing by following at least those specified in homework assignments. Those who fail to follow these rules may get at most 10 points off.
- The maximum length of the report is ***ten pages***, including everything. Everything starting from page 11 will not be graded.

2 The story

As one of the most famous food manufacturing companies in Taiwan, the IEDO company produces and sells sausage, ham, jerky, and other products.[1] Due to the huge demand volume, whether its factory is operated efficiently has significant impact on the company's profitability.

Almost all the products made by IEDO must go through *the smoking station*, which consists of five machines (called "smokers" in many places[2]) and opens at 7:30 AM. There are three types of processes that may happen in a smoking station: *boiling*, *baking*, and *smoking*. Different products have different sequences of processes. For example, to allow a typical set of bacon leave the smoking station, we need 100 minutes of baking, 25 minutes of smoking, 30 minutes of boiling, another 60 minutes of boiling, and 150 minutes of baking. Each of the other products has its own sequence of processes, and the amounts of time vary according to the production amount. All the processes must be completed in order, and a process may be started only after its previous process has been completed.

In each morning, you need to schedule several *jobs* to the five machines, where a job is to manufacture one product for one customer. Interestingly, while there are five machines in the smoking station, machine 1, which is the oldest and cheapest, can only do the boiling process. On the contrary, each of machines 2 to 5 are able to conduct all the three types of processes. To complete the jobs as soon as possible, you should not abandon machine 1. To utilize machine 1, a job may be split into two pieces so that the two pieces are completed by two different machines (one is machine 1 and the other is one of machines 2 to 5). Job splitting obviously complicates the scheduling task. Luckily, due to the complicated nature of food manufacturing, a job cannot be split arbitrarily. For each job, at most one *splitting timing* is predetermined, which specifies exactly after which process may a job be split. Note that it is possible that some jobs cannot be split. If a job's splitting timing is 0, the job cannot be split.

Let's make things clearer with the example provided in Table 1. In this example

[1] This case study is a simplification of a real industry project did by the instructor and the Black Bridge Food Co. in the past. The company's website is at `https://www.blackbridge.com.tw/`. If you may solve it, you are quite close to using Operations Research to solve real-world business optimization problems!

[2] To see some pictures of smokers, you may search for it online. One example may be found at `https://www.maurer-atmos.de/kochen/`.

instance, which was something happened in a past day, there are twelve jobs to be done. These jobs may be categorized into several groups:

Job	Process Type						S.T.
	Process 1	Process 2	Process 3	Process 4	Process 5	Process 6	
1	Boiling	Baking	Smoking				2
2	Boiling	Baking	Boiling				1
3	Boiling	Baking	Boiling	Baking	Boiling		1
4	Smoking	Boiling	Baking	Boiling	Baking	Smoking	3
5	Boiling	Baking	Boiling				2
6	Boiling						0
7	Baking	Smoking					1
8	Baking	Smoking					0
9	Boiling	Baking	Boiling				2
10	Baking	Boiling	Baking	Boiling	Baking		2
11	Boiling	Baking	Boiling				2
12	Boiling						0

Table 1: Process types of a example instance (S.T. means splitting timing)

- Jobs 6 and 12 have only one process. Their splitting timings are thus naturally 0.
- Job 8 has two processes, first baking and then smoking. However, its special characteristic makes it impossible to be split. It therefore should be scheduled to only one machine (and obviously it cannot be scheduled on machine 1).
- Jobs 1, 4, 7, and 10 all have multiple processes and may be split. For example, job 1 requires three processes, which are boiling, baking, and smoking. As its splitting timing is 2, job 1 may be split into two pieces, where the first piece contains the first two processes and the second piece contains the third process. Nevertheless, note that neither the first piece nor the second piece may be done by machine 1. Only machines 2 to 5 may be used to complete job 1. As another example, job 4 has six processes and may be split into two pieces, each having three processes. None of the two pieces may be done by machine 1.
- Jobs 2, 3, 5, 9, and 11 all have multiple processes and may be split. Moreover, one of the two pieces contains only boiling and may be done by machine 1. For example, job 3 may be split into two pieces where the first piece contains only one

process of boiling. As machine 1 can do boiling, the first piece of job 3, if job is split, may be done on machine 1. The second piece may then be started on one of machines 2 to 5 after machine 1 finishes the first piece.

To make a schedule, certainly more information is needed. Table 2 lists the processing time of each process of each job and due time of each job. For example, there are three processes in job 1, each requiring 2.7 hours, 1 hour, and 0.5 hour. If job 1 is scheduled on a single machine starting at 7:30 AM, it will take $2.7 + 1 + 0.5 = 4.2$ hours to complete job 1. Its completion time will be 11:42 AM, 4.2 hours after 7:30 AM. As its due time is 12:30 AM, job 1 meets the due time and is not tardy.

Job	Processing Time (in Hours)						D.T.
	Process 1	Process 2	Process 3	Process 4	Process 5	Process 6	
1	2.7	1	0.5				12:30
2	1.6	1.4	0.9				12:30
3	1	0.9	0.2	0.2	1.4		12:30
4	0.5	0.7	1	0.6	0.3	0.5	12:30
5	0.8	1	0.9				12:30
6	2.7						12:30
7	1.4	1.5					17:30
8	1.1	1.1					17:30
9	0.8	1	0.8				17:30
10	1	0.5	0.7	1.1	1.1		17:30
11	1.1	1.4	1.5				17:30
12	2.3						17:30

Table 2: More information of the example instance (D.T. means due time)

The Gantt chart of an example complete schedule for the above instance is depicted in Figure 1. In this schedule, job 2 is split so that its boiling process is on machine 1 and its baking and second boiling processes are on machine 2. Jobs 5, 9, and 11 are also split so that machine 1 may be utilized. All the jobs are completed around 18:00.

There are two objectives of your scheduling task. First, the *total tardiness* should be minimized. According to Figure 2, jobs 1 and 10 are tardy, and the total tardiness is $1.9+0.3 = 2.2$ hours (precise calculation may be done using numbers in Table 2). Second, the *makespan*, which is latest completion time among all jobs, should be minimized. According to Figure 2 again, the makespan is around 18:00. A better schedule is depicted

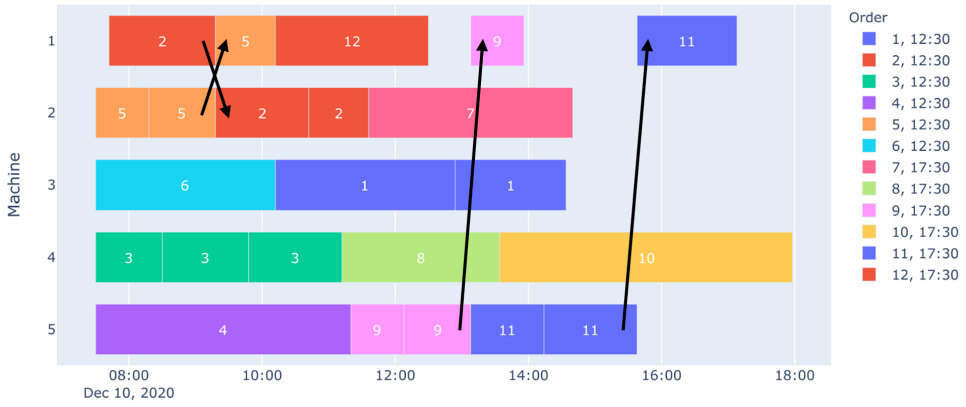

Figure 1: A (bad) schedule for the example instance

in Figure 3. In this better schedule, there is no tardy job, and the makespan is slightly before 17:30. The first objective has higher priority than the second one. In other words, when comparing two schedules, the one with the lower total tardiness is considered better regardless of their makespans. Their makespans are compared and the smaller one is preferred only when their total tardiness are identical.

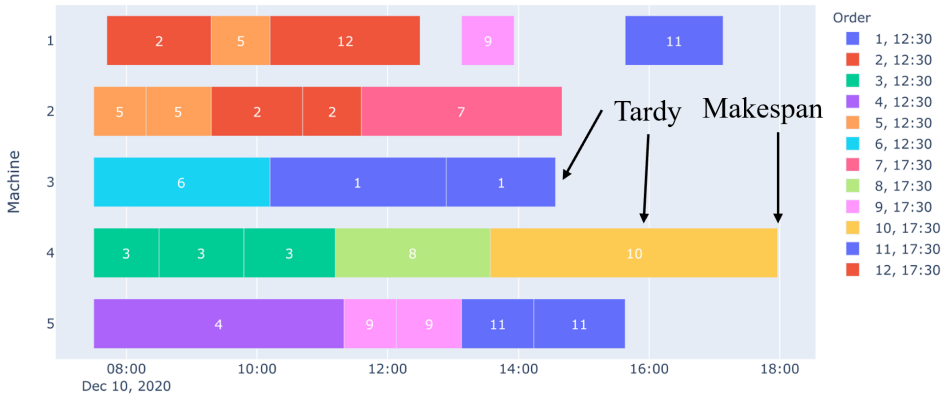

Figure 2: Illustration of the objective functions

Once a piece of a job is started on a machine, it cannot be stopped until the whole piece has been completed. If a job has two pieces, however, there may be a time gap between the completion time of the first piece and the starting time of the second piece even if the two pieces are on the same machine. Nevertheless, foods under processing should not be exposed in air for too long. Therefore, the inter-processing time is not allowed to exceed one hour.

Given all the information above, please try to help the IEDO company solve the scheduling problem.

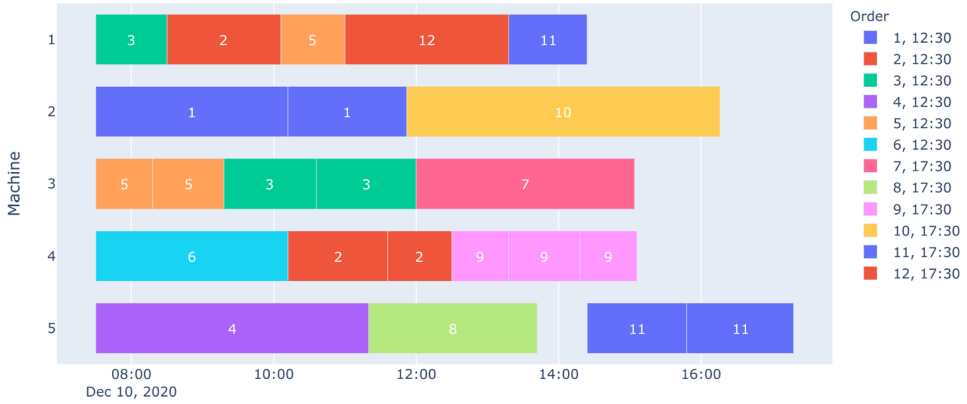

Figure 3: A (good) schedule for the example instance

3 Abstraction, generalization, and instance files

To solve IEDO's scheduling problem regarding food processing, we may formulate mathematical programs and invoke commercial solvers like Gurobi Optimizer to look for a good schedule. While this is good, and we will ask you to do that, too large instances cannot be solved by integer programming, and a heuristic algorithm is needed in this case. Therefore, you will also be invited (actually forced) to design and implement a heuristic algorithm for the scheduling problem. Your implementation will be graded based on its performance (i.e., objective values), efficiency (i.e., execution time), and design (depending on how you convince the TAs that your algorithm should be good).

Before we describe your tasks in this project, let's first introduce the concepts of *abstraction* and *generalization* for you. Understanding the concepts will also help you understand the format of input instance files.

Along with this document, five CSV files are given to you. Each CSV file represents an instance. Once you take a look at any of the CSV files, you will realize that the given instances are *abstraction* of the original concrete instances: Details that are irrelevant to the solution processes are ignored. Let's take the example specified in Section 2 (which is not those instances in any of the given CSV files) as an example. While the concrete information of that instance was provided in Tables 1 and 2, now it is condensed into Table 3 below. Each instance in an CSV file follows the same six-column format.

The original concrete instances are condensed into the six-column format in the following way:

- In the new format, each row (except the first one) contains the information of a

Job ID	Stage-1 P.T.	Stage-2 P.T.	Stage-1 M.	Stage-2 M.	Due Time
1	3.7	0.5	2,3,4,5	2,3,4,5	5
2	1.6	2.3	1,2,3,4,5	2,3,4,5	5
3	1	2.7	1,2,3,4,5	2,3,4,5	5
4	2.2	1.4	2,3,4,5	2,3,4,5	5
5	1.8	0.9	2,3,4,5	1,2,3,4,5	5
6	2.7	0	1,2,3,4,5	N/A	5
7	1.4	1.5	2,3,4,5	2,3,4,5	10
8	1.1	1.1	2,3,4,5	2,3,4,5	10
9	1.8	0.8	2,3,4,5	1,2,3,4,5	10
10	1.5	2.9	2,3,4,5	2,3,4,5	10
11	2.5	1.5	2,3,4,5	1,2,3,4,5	10
12	2.3	0	1,2,3,4,5	N/A	10

Table 3: An example instance (P.T. means processing time, and M. means machines)

job. In particular, the column "Job ID" records the IDs of jobs. All values in this column are positive integers in $\{1, 2, ..., 1000\}$.

- We now do not care whether the factory opens at 7:30, 8:00, or 11:00. We will label the opening time as time 0. Each job has a due time, which is now labeled as the number of hours (may be fractional in general) after the opening time. In the above example instance, the (abstract) due time of job 1 is recorded as 5 because its (concrete) due time is 12:30 and the factory opening time is 7:30.

  The (abstract) due times are recorded in the column "Due Time". All values in this column are positive real numbers in $(0, 24]$.

- We now do not care whether a job has one process, two processes, or ten processes. As each job may be split at most once, we only care about whether it has one stage or two stages. For example, jobs 6 and 12 have only one stages because they cannot be split. For a job that cannot be split, we define its (abstract) stage-1 processing time as its (concrete) total processing time and its (abstract) stage-2 processing time as 0. On the contrary, the other ten jobs all have two stages because they may be split once. For a job that can be split, we define its (abstract) stage-1 processing time as the sum of (concrete) processing times before the splitting time and its (abstract) stage-2 processing time as the sum of the remaining (concrete) processing times. As an example, the (abstract) stage-1 processing time of job 4 is $0.5 + 0.7 + 1 = 2.2$, and that of stage 2 is $0.6 + 0.3 + 0.5 = 1.4$.

The (abstract) processing times are recorded in the columns "Stage-1 Processing Time" and "Stage-2 Processing Time". All these values are real numbers in $(0, 10]$.

- Recall that in the practical situation faced by the IEDO company, machine 1 can only do boiling while machines 2 to 5 may do all three types of processes. We now do not care about any concrete process type. We also do not care whether it is machine 1 or machines 1 and 2 that cannot do a specific type of process. All we need, from the perspective of solving the problem, is to know for each stage of each job the list of machines that may process that stage. Let's now take job 2 as an example. Recall that job 2's first stage contains only one process, which is boiling, and its second stage contains two processes, baking and boiling. Therefore, its first stage may be processed by machines 1, 2, 3, 4, and 5, but its second stage may only be processed by machines 2, 3, 4, and 5.

  The lists of allowable machines are recorded in the columns "Stage-1 Machines" and "Stage-2 Machines". There are two possible types of values in these two columns. If a job does not have stage 2, its value in the "Stage-2 Machines" column will be "N/A" (without quotation marks). Otherwise, a list of machines will be a string of comma-separated numbers inside a pair of double quotation marks, where numbers are all valid machine IDs, no two numbers are identical, and no white space exists.

- The range of the number of machines should also be specified. In this project, the number of machines is an integer between 1 and 20 (including 1 and 20). For each instance, the information regarding the number of machines may be obtained by processing the lists of allowable machines.

To make the problem slightly easier, let's assume the inter-processing time limit is always one hour throughout this project.

Abstraction is important in problem solving. In many cases, different problems become the same one after abstraction, and designing a method that solve an abstract problem actually solves all those concrete problems.[3] Writing compact formulations of mathematical programming is one great way of abstraction. In the sequel, we will introduce your tasks according to the abstract problem described above.

To help you make sure that you have correct understanding of these instances, here we also provide the objective values of the optimal solutions. The minimized total tardiness

[3]This is also called "reduction" computer science.

are 2.20, 0.00, 2.45, 2.27, and 2.30, and the minimized makespans (when the total tardiness are set to the minimum levels) are 5.60, 7.60, 17.18, 20.09, and 9.00 for instances 1, 2, 3, 4, and 5, respectively.

4 Your tasks

Problem 1

(15 points) Please formulate *two* mathematical program (which ideally should be a linear integer program) that solves the *generalized* problem described in Section 3.[4] Each instance will be solved in two stages, one with a program. In the first stage, we minimize total tardiness with the first program and obtain the minimum total tardiness that is attainable. In the second stage, we then minimize makespan with an additional constraint requiring the total tardiness to be the minimum level attained in the first stage. In your report, write down your mathematical models using rigorous mathematical notations and expressions.

Problem 2

(15 points) Please write a computer program to invoke an optimization library that may solve instance 5, which is given in the file instance05.csv. We suggest you to write a Python 3.9 program to invoke Gurobi Optimizer. It is fine if you prefer something else, but if you do so, please teach yourself and do not ask the instructing team when you encounter difficulties.

Do not submit your computer program. Instead, submit an optimal schedule you obtain by solving instance 5 by telling us how to split jobs and how to assign jobs (or pieces of jobs) to machines. Report the objective value at the same time. Your schedule must be complete and precise so that those employees in the IEDO company may execute your schedule. The way you describe your schedule should also be clear and easy to read so that non-technical people may understand. In short, it should be in a business language.

[4] In other words, the number of machines is not limited to 5, not only machine 1 is not fully functional, there can be more than two distinct due time timings, etc.

Problem 3

(30 points) When the scale of an instance is large (e.g., with hundreds of orders and more than ten machines), it may be too time-consuming to look for an optimal solution. Therefore, please implement a heuristic algorithm in Python 3.9 so that the instructing team may grade it with fifteen testing instances, one in an CSV file. Five of these CSV files have been provided to you for your reference. You will not see the other ten hidden CSV files until this assignment is due. However, you may assume that they follow the same format described in Section 3. Some of these hidden instances are of large scales so that Gurobi Optimizer cannot report an optimal solution easily.

Please note again that in these fifteen instances, the number of machines may not be 5, functionality of these machines may vary a lot, there may be more than two distinct timings for due times, etc. In short, you are now dealing with the generalized problem described in Section 3. There is a saying "abstraction is the foundation for generalization." Please try to think about this.

You are also provided three PY files: algorithm_module.py, grading_program.py, and MTP_lib.py. You should implement your algorithm in algorithm_module.py, which imports libraries that are allowed to be used from MTP_lib.py. DO NOT change the function name and file name. Moreover, please make sure that this function returns a schedule specified in two two-dimensional lists `machine` and `completion_time`. See the comments in the two PY files for more information regarding the two lists. The instructing team will use grading_program.py to invoke the function `heuristic_algorithm` in the file algorithm_module.py you submit to obtain `machine` and `completion_time` returned by your heuristic algorithm. The function `check_format` will then be invoked to check the format of the returned values, and a function `find_obj_value` (which will not be provided to you) will then be invoked to check the feasibility of your plan and calculate the objective value for your plan (if it is feasible). You may check out some more instructions in these PY files.

One team may earn at most 2 points for each of the fifteen instances (including the five whose contents have been revealed to you). First, one team earns 1 point if the generated schedule is feasible (no two stages overlap with each other on a single machine, a stage is not processed by a machine that cannot process that stage, the inter-processing times of all jobs are no greater than one hour, etc.) and its execution time is no longer than three minutes. If the schedule is infeasible or the execution time is longer than three minutes, the team earns 0 points in this category. Please note that your implementation

will be executed on a TA's device. The device will not be a very slow one, but to make sure that your program may return a solution in three minutes, you may want to have some buffer on your own device.

Another 1 points are based on performance. One team earns all or part of the other 1 points according to her/his ranking among all teams. Let r_i be the ranking of team i, where $r_i = 1$ means team i's schedule is the best, team i earns

$$1 - \frac{r_i - 1}{N - 1}$$

points, where N is the number of teams getting a feasible schedule for this instance. Two schedules are compared according to the rule specified in Section 2. In particular, we prefer a schedule with lower total tardiness. If the total tardiness are the same, we prefer the one with lower makespan. The rankings for different instances are independent. For example, one team may be ranked 1 in instance 1 but ranked 5 in instance 2.

Important note. There is no restriction on the design of your heuristic algorithm. However, we require you to implement your heuristic algorithm in Python 3.9. We understand that it is better if there is no restriction on the programming language. However, as we need to specify the input/output to give you a precise development instruction, facilitate grading, and use the same basis to time all submissions, we are sorry that we must make such a restriction.

In your Python 3.9 program, feel free to use the following libraries: `gurobipy`, `math`, `numpy`, `pandas`, `time`, `os`, `itertools`, `random`, `copy`, and `csv`. Regarding `pandas`, note that version 2.0.0 has been released and has several significant differences from previous versions (some functions are even removed). As version 2.0.0 will be used by the TAs, if you want to use `pandas`, please make sure that you are also using version 2.0.0. We believe these libraries are quite enough and do not expect/suggest one to use other libraries.

If you really need to use some other libraries (which we do not suggest), you need to reply to a post on NTU COOL to obtain the permission from the TAs (typically the TAs will say yes as long as they may install the libraries you need in their Python 3.9 environment, but we do not guarantee that the TAs will accept all requested libraries). Once the TAs respond to your message and say yes, they will add that library (those libraries) into their MTP_lib.py so that your program will be graded accordingly. However, a student cannot ask the TAs to try to run her/his program before this midterm project is due.

Problem 4

(20 points) Use your own words to introduce your heuristic algorithm to the instructing team. You may write paragraphs of words, draw flow charts, present pseudocodes, and/or provide examples. There should also be a basic big-O analysis regarding the time complexity of your algorithm. Points will be given to you according to the logic of your algorithm and clearness of your description. Please try your best to convince the instructing team that your algorithm design is intuitively good.

Problem 5

(20 points) Suppose that we designed and implemented a heuristic algorithm in Problem 2. Even if we tried to solve instances 1 to 5 to test the performance of our heuristic algorithm, how may we be confident in the performance for solving remaining instances? Only five instances seem to be insufficient!

To make ourselves more confident, we may want to conduct some numerical experiments before we submit our heuristic algorithm (to the instructing team or to an organization that will use our heuristic algorithm in practice). To do so, we write another program to generate random instances, let our algorithm solves these random instances, and compare the objective values generated by our algorithm with some benchmark.

Designing factors and scenarios. Regarding generating random instances, please make your own decisions in choosing fixed values for some of the parameters and probability distributions for some other parameters. Below are several examples settings to generate random instances. All instances generated by a single setting is said to be in the same *scenario*:

1. Let the number of jobs be fixed at 25 and the number of machines be fixed at 5. Each job has 30% of probability to have only one stage and 70% to have two stages. For each piece of each job, the processing time is uniformly drawn from $[1, 10]$. For each job, the first piece can be processed by all machines, and the second piece can only be processed by machines 2 to 5. For each job, the due time is randomly assigned to be 5 or 10 with equal probability.

2. All jobs have two stages. All the other settings are identical to scenario 1.

3. For each piece of each job, each machine is allowed to process it with probability

50%. All the other settings are identical to scenario 1.

4. Let the number of jobs be fixed at 50 and the number of machines be fixed at 10. All the other settings are identical to scenario 1.

5. For each job, the due time is randomly assigned to be 4 or 8 with equal probability. All the other settings are identical to scenario 1.

The above five settings create five different scenarios, and we may generate hundreds or thousands of instances for each scenario to test our algorithm. By comparing scenario 1 with each of scenarios 2, 3, 4, and 5, we may test the behavior and performance of our algorithm along different *factors*: complexity of jobs, machine capability, scale of instances, and tightness of capacity. If our algorithm performs well in all scenarios, we will be confident about it. The confidence should be much higher than testing it with only one scenario. Of course, the above five scenarios are just examples. It is possible that they are not appropriate, and you may choose your own way to design your scenarios.

Ideally, your random instances should cover a wide range of situations. However, if you find that your algorithm cannot deal with some weird cases, it is fine to narrow down the ranges of your parameter values as long as you clearly specify the experiment setting. Do not test your algorithm with just one or two instances. Try to test it with hundreds or thousands of instances.

Benchmarks. Regarding benchmarks, one typical candidate is your mathematical program constructed in Problem 1. As long as your instances are not too big, you may use a solver to generate an optimal solution and compare your heuristic solution with the optimal one. When your instances are big, another typical candidate is the relaxation of your integer program (which may or may not be appropriate). You may take away the integer constraint or some other constraints to find an upper bound or lower bound of the objective value of an optimal solution. It is also common for one to compare the proposed algorithm with some simple (or even stupid) ones. After all, if the proposed algorithm cannot outperform a simple one, it cannot be good.

A complete example. To get a more concrete idea about how to conduct experiments to analyze the performance of your heuristic algorithm, you are suggested to take a look at the paper "fairAllocation_230822.pdf" provided to you with this document. You may go through the first paragraph of Section 3 very quickly to understand the problem, and then jump to Sections 5.1, 5.2, and 5.4 to see an example about designing factors and scenarios, generating instances, designing benchmarks, run experiments, and describe all of these. There are something that you do not need to do:

- While in Section 5.2 of that paper you see a paragraph describing the genetic algorithm, you are not required to do this. When benchmarking your heuristic algorithm, a very simple algorithm is enough.
- While in Section 5.3 of that paper you see another version of the problem, you do not need to have multiple versions. Then of course you do not need to worry about the performance of your heuristic algorithm in such an another version, even though this is done in Section 5.4 of that paper.
- While in Section 5.5 of that paper you see some analysis regarding computation time, you are not required to do this.

Your task. In this problem, please follow the above instructions to conduct your own experiments to convince the instructing team that your algorithm performs well. Clearly indicate how the instances are generated by writing down those fixed values and probability distributions. If you have multiple scenarios, using a table to list them should be a good idea. Specify the benchmarks you adopt, and if it is a simple heuristic algorithm, indicate its steps. For each instance, record the gap between your heuristic solution and an optimal one (to the original problem or a relaxed problem), the gap between your heuristic solution and that generated by another simple heuristic algorithm, etc. Write down the average and standard deviation and draw a histogram for those gaps to (hopefully) show that (1) the performance of your heuristic algorithm is close to an optimal solution, and (2) the performance of your heuristic algorithm is much better than those benchmark simple heuristic algorithms.

Finally, keep this in mind: Designing an algorithm is not hard; making it convincing is. Please do everything you may do to make your proposed algorithm convincing.