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

Team O

Hong-Kai Yang B11611047
Yi-Ting Chen B11705051
Chi-Wei Ho B11702044
Yu-Ting Chou B11702080

May 4, 2024

Problem 1

Let $J = \{1, ..., \text{n}\}$ be the set of jobs, $S = \{1, 2\}$ be the stage of job, and $K = \{1, ..., \text{m}\}$ be the set of machines.

Let c_{is} be the completion time of job $i - s$ $\forall i \in J$, $\forall s \in S$; the processing time of job $i - s$ be p_{is} $\forall i \in J$, $\forall s \in S$, job i's due time and tardiness be d_i and t_i $\forall i \in J$, M be the sum of the processing time of all jobs on stage 1 and stage 2, which is $M = \sum_{\forall i \in J} \sum_{\forall s \in S} p_{is}$.

Let $y_{mis} = 1$, if job $i - s$ $\forall i \in J$, $\forall s \in S$ is made by machine $m \in K$; 0 otherwise.

Let $A_{mis} = 1$, if job $i - s$ $\forall i \in J$, $\forall s \in S$ can be made by machine $m \in K$; 0 otherwise.

Let $z_{misjt} = 1$, if job $i - s$ $\forall i \in J$, $\forall s \in S$ and job $j - t$ $\forall j \in J$, $\forall t \in S$ are both made by machine $m \in K$, and job $i - s$ $\forall i \in J$, $\forall s \in S$ is proceeded before job $j - t$ $\forall j \in J$, $\forall t \in S$; 0 otherwise.

In the first stage, our objective is to minimize the total tardiness. Here is our formulation:

$$
\begin{aligned}
\min \quad & \sum_{i \in J} t_i = t^* \\
\text{s.t.} \quad & t_i \geq 0 \quad \forall i \in J \\
& t_i \geq c_{i,2} - d_i \quad \forall i \in J \\
& \sum_{m \in K} y_{mis} = 1 \quad \forall i \in J, \quad \forall s \in S \\
& y_{mis} \leq A_{mis} \quad \forall m \in K, \quad \forall i \in J, \quad \forall s \in S \\
& y_{mis} + y_{mjt} \leq z_{misjt} + z_{mjtis} + 1 \quad \forall m \in K, \quad \forall i \in J, \quad \forall j \in J, \quad \forall s \in S, \quad \forall t \in S, \quad (i,s) \neq (j,t) \\
& c_{is} - (c_{jt} - p_{jt}) \leq M\,(1 - z_{misjt}) \quad \forall m \in K, \quad \forall i \in J, \quad \forall j \in J, \quad \forall s \in S, \quad \forall t \in S \quad (i,s) \neq (j,t) \\
& c_{i,2} - p_{i,2} \geq c_{i,1} \quad \forall i \in J \\
& c_{i,2} - p_{i,2} - c_{i,1} \leq 1 \quad \forall i \in J \\
& c_{is} \geq p_{is} \quad \forall i \in J, \quad \forall s \in S \\
& c_{is} \geq 0 \quad \forall i \in J, \quad \forall s \in S \\
& y_{mis}, A_{m_{is}} \in \{1, 0\} \quad \forall m \in K, \quad \forall i \in J, \quad \forall s \in S \\
& z_{misjt} \in \{1, 0\} \quad \forall m \in K, \quad \forall i \in J, \quad \forall s \in S, \quad \forall j \in J, \quad \forall t \in S.
\end{aligned}
$$

In the second stage, our objective is to minimize makespan with constraints requiring the total tardiness to be the minimum level attained in the first stage. Here is our formulation:

$$
\begin{aligned}
\min \quad & w \\
\text{s.t.} \quad & w \geq c_{i,2} \\
& \sum_{i \in 1} t_i = t^* \\
& t_i \geq 0 \quad \forall i \in J \\
& t_i \geq c_{i,2} - d_i \quad \forall i \in J \\
& \sum_{m \in K} y_{mis} = 1 \quad \forall i \in J, \quad \forall s \in S \\
& y_{mis} \leq A_{mis} \quad \forall m \in K, \quad \forall i \in J, \quad \forall s \in S \\
& y_{mis} + y_{mjt} \leq z_{misjt} + z_{mjtis} + 1 \quad \forall m \in K, \quad \forall i \in J, \quad \forall j \in J, \quad \forall s \in S, \quad \forall t \in S, \quad (i,s) \neq (j,t) \\
& c_{is} - (c_{jt} - p_{jt}) \leq M\,(1 - z_{misjt}) \quad \forall m \in K, \quad \forall i \in J, \quad \forall j \in J, \quad \forall s \in S, \quad \forall t \in S \quad (i,s) \neq (j,t) \\
& c_{i,2} - p_{i,2} \geq c_{i,1} \quad \forall i \in J \\
& c_{i,2} - p_{i,2} - c_{i,1} \leq 1 \quad \forall i \in J \\
& c_{is} \geq p_{is} \quad \forall i \in J, \quad \forall s \in S \\
& c_{is} \geq 0 \quad \forall i \in J, \quad \forall s \in S \\
& y_{mis}, A_{m_{is}} \in \{1, 0\} \quad \forall m \in K, \quad \forall i \in J, \quad \forall s \in S \\
& z_{misjt} \in \{1, 0\} \quad \forall m \in K, \quad \forall i \in J, \quad \forall s \in S, \quad \forall j \in J, \quad \forall t \in S.
\end{aligned}
$$

Problem 2

There will be five machines processing ten jobs, and the schedule are as follows:

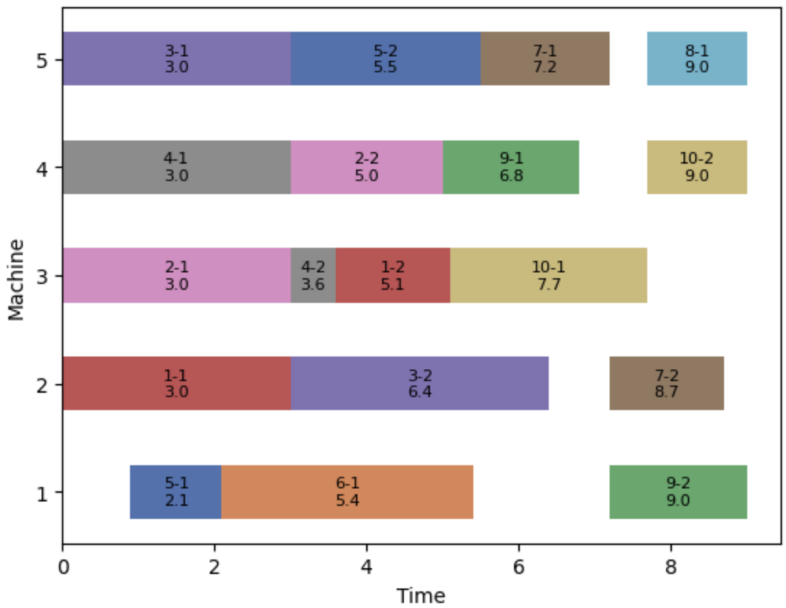

Figure 1: Scheduling Chart

Machines	Schedules
Machine 1	Machine 1 starts at 0.9, processes job 5, stage 1 for 1.2 units, finishing at 2.1. Then it processes job 6, stage 1 for 3.3 units, finishing at 5.4. After a wait of 1.8 units, it processes job 9, stage 2 for 1.8 units, finishing at time 9.
Machine 2	Machine 2 starts at 0, processes job 1, stage 1 for 3 units, finishing at 3. Then it processes job 3, stage 2 for 3.4 units, finishing at 6.4. After a wait of 0.8 units, it processes job 7, stage 2 for 1.5 units, finishing at time 8.7.
Machine 3	Machine 3 starts at 0, processes job 2, stage 1 for 3 units, finishing at 3. Then it processes job 4, stage 2 for 0.6 units, finishing at 3.6. Next, it processes job 1, stage 2 for 1.5 units, finishing at 5.1. Finally, it processes job 10, stage 1 for 2.6 units, finishing at 7.7.
Machine 4	Machine 4 starts at 0, processes job 4, stage 1 for 3 units, finishing at 3. Then it processes job 2, stage 2 for 2 units, finishing at 5. Next, it processes job 9, stage 1 for 1.8 units, finishing at 6.8. After a wait of 0.9 units, it processes job 10, stage 2 for 1.3 units, finishing at time 9.
Machine 5	Machine 5 starts at 0, processes job 3, stage 1 for 3 units, finishing at 3. Then it processes job 5, stage 2 for 2.5 units, finishing at 5.5. Next, it processes job 7, stage 1 for 1.7 units, finishing at 7.2. After a wait of 0.5 units, it processes job 8, stage 1 for 1.3 units, finishing at time 9.

Problem 4

By default, iterations using for i in range start from an index of 0. However, for the sake of clarity and readability, the range in our pseudocodes begins with an index of 1, adhering to a one-based indexing approach.

After reading the files into the program, we use an array called "jobs" to store all the tasks and sort them by urgency using binary search. Urgency is defined as "Due – s1 – s2," representing the amount of time remaining before a job becomes overdue, assuming the job starts immediately. A negative value indicates that the job will already be overdue if it starts immediately.

Explanation of heuristic algorithm

```
Define Class JOB include (id, s1, s2, m1, m2, due, lefttime, s1_finishtime)
function JOB(id, s1, s2, m1, m2, due)
    self.id ← id
    self.s1 ← s1
    self.s2 ← s2
    self.m1 ← m1
    self.m2 ← m2
    self.due ← due
    lefttime ← 0
    s1_finishtime ← 0
end function
```

Figure 2: Defining Class Job

Job: A custom class containing the attributes id, s1, s2, m1, m2, due, leftime, s1.finishtime. These represent the job ID, processing time for Stage One, processing time for Stage Two, the machines available for Stage One and Stage Two, the job's due time, lefttime is the total left time of the entire job, and the s1 finishtime is the time when the job's stage 1 is finished, respectively.

```
if len(jobs) ≠ 0 then
    for i from 1 to len(jobs) do
        if jobs[i].s1_finishtime = 0 and machine_id in jobs[i].m1 then
            if (not THEEARLIESTMACHINE(Job.m1)) then
                return tardiness
            else if jobs[i].s2 ≠ 0 and PLAN(jobs[i], machine_time[machine_id],
machine_time) then
                mt ← FINDEARLISTMACHINE(jobs[i].m2, machine_time)
                for j from 1 to len(jobs) do
                    if i ≠ j and jobs[j].s2 == 0 and
machine_time[machine_id]+jobs[j].s1 ≤ mt − jobs[i].s1 then
                        machine_time[machine_id] += jobs[j].s1
                        tardiness+ = max(0, machine_time[machine_id]-jobs[j].due)
                        jobs.pop(j)
                        return tardiness
                    end if
                end for
                machine_time[machine_id] = mt
                continue
            end if
            jobs[i].lefttime− = jobs[i].s1
            machine_time[machine_id] += jobs[i].s1
            jobs[i].s1_finishtime = machine_time[machine_id]
            if jobs[i].lefttime = 0 or jobs[i].s2 == 0 then
                tardiness+ = max(0, machine_time[machine_id]-jobs[i].due)
                jobs.pop(i)
                return tardiness
            end if
        end if
```

Figure 3: Defining Pick Job-1

The following sections are the explanation of the function pick job:

First, the initial "if" statement is used to assign a machine to Stage 1 of the job while simultaneously using the function plan to assess whether the arrangement is feasible and determine if adjustments are necessary. After Stage 1 is scheduled, it will be checked whether there is a Stage 2 for the job. If there isn't, the job will be removed from the queue, indicating that it has been completed. A detailed description of the function plan will be elaborated in the following.

```
    if machine_time[machine_id] - jobs[i].s1_finishtime ≤ 1 and jobs[i].lefttime − jobs[i].s2 == 0 and machine_id in jobs[i].m2 then
        if machine_time[machine_id] ≥ jobs[i].s1_finishtime then
            machine_time[machine_id] += jobs[i].s2
            tardiness+ = max(0, machine_time[machine_id]-jobs[i].due)
            jobs.pop(i)
            return tardiness
        end if
```

Figure 4: Defining Pick Job-2

The second "if" checks if the same machine cannot continue the job. If it can, the job is scheduled to minimize wait time. Once Stage 2 is completed, the job is removed from the work queue and tardiness is calculated.

```
else if jobs[i].s1_finishtime > 0 and jobs[i].s2 ≠ 0 then
    for id in chooseSequence do
        if id not in jobs[i].m2 then
            continue
        end if
        for j from 1 to len(jobs) do
            if j == i ormachine_id not in (jobs[j].m1&jobs[j].m2) then
                continue
            end if
            if jobs[j].s2 ≠ 0 and machine_time[id] + jobs[j].s1 + jobs[j].s2 ≤ jobs[i].s1_finishtime then
                machine_time[id] += jobs[j].s1 + jobs[j].s2
                tardiness+ = max(0, machine_time[id] − jobs[j].due)
                jobs.pop(j)
                return tardiness
            end if
        end for
```

Figure 5: Defining Pick Job-3

After the initial "if" statement, if the same machine cannot continue processing the job, another machine must be assigned for Stage 2. A suitable machine is ensured because the function plan checks and adjusts for feasibility. The selected machine's current makespan will be less than the completion time of stage 1 plus a one-hour buffer, although the difference in time may vary. If the gap is sufficiently large to accommodate another job that the machine can handle, it should be prioritized to fill the idle time.

The initial loop uses the function chooseSequence (which presorts machines) to iterate through available jobs, searching for jobs that can be processed by the current machine and share both stage 1 and stage 2. If the conditions are satisfied and the job can be feasibly scheduled, it is added to fill the idle period. After scheduling, the job is removed from the work queue and tardiness is calculated.

```
for j from 1 to len(jobs) do
    if i ≠ j andjobs[j].s2 == 0 and machine_time[id] + jobs[j].s1 ≤ jobs[i].s1_finishtime then
        machine_time[id]+ = jobs[j].s1
        tardiness+ = max(0, machine_time[id] − jobs[j].due)
        jobs.pop(j)
        return tardiness
    end if
end for
```

Figure 6: Defining Pick Job-4

We find it more efficient to prioritize scheduling tasks that can be processed in both stages before considering tasks that can only be handled on Stage 1 machines. In the same manner, the second loop schedules tasks exclusively handled by Stage 1 machines during idle periods. Once scheduling is complete, the task is removed from the list and tardiness is calculated.

```
                machine_time[id]+ = jobs[i].s2
                tardiness+ = max(0, machine_time[id] − jobs[i].due)
                jobs.pop(i)
            end for
        end if
    end for
end if
return tardiness
```

Figure 7: Defining Pick Job-5

After scheduling tasks for Stage 2, remove them from the task list and calculate the tardiness.

```
function PLAN(job, current_machine_time, machine_time)
    plan_job ← job.copy()
    current_machine_time += plan_job.s1
    plan_job.s1_finishtime ← current_machine_time
    choice ← job.m2
    machine_list ← []
    for index in choice do
        index ← int(index)
        machine_list.append(machine_time[index − 1])
    end for
    for mt in machine_list do
        if mt == 0 or mt < plan_job.s1_finishtime + 1 then
            return True
        end if
    end for
    return False
end function
```

Figure 8: Finding Function Plan

Plan: This function is executed only when there is a Stage 2 task available. Its purpose is to ensure that scheduling a job will remain feasible after execution, preventing subsequent complications.

The process works as follows:

Check Feasibility: Evaluate whether, after scheduling a machine to complete the job, another machine can take over the task within an hour of the job's completion to avoid missing the due date. If feasible, return True, allowing us to proceed with scheduling Stage 1 of the job on the specified machine.

Return False if Adjustment Needed: If no machine is available to continue the task, return False, indicating the need for adjustments.
The "plan" function only assesses whether adjustments are necessary. The adjustments themselves occur within an if statement after the plan function. These adjustments are required when a machine cannot handle both Stage 1 and Stage 2 sequentially. In some cases, other machines may already be heavily occupied, necessitating a delay in starting the current job so that another machine can pick up Stage 2 within the required timeframe.

When a job is delayed, there may be idle time before the task begins. In such cases, other tasks that can be handled by the same machine and involve only Stage 1 processing can be scheduled during the idle period to optimize machine usage.

```
function TheEarliestMachine(job, machine_time, machine_id)
    if machine_time[machine_id] == 0 then
        return True
    end if
    choice ← job.m1
    machine_list ← []
    for index in choice do
        machine_list.append(machine_time[index])
    end for
    best ← min(machine_list)
    if machine_time[machine_id] == best then
        return True
    end if
    return False
end function
function FindEarliestMachine(job, machine_time)
    choice ← job.m2
    machine_list ← []
    for index in choice do
        machine_list.append(machine_time[index])
    end for
    for elem in machine_list do
        if elem == min(machine_list) then
            return elem
        end if
    end for
end function
```

Figure 9: Defining FindEarliestMachine and TheEarliestMachine Function

The function FindEarliestMachine supports subsequent operations within the plan function. Its purpose is to identify the earliest possible start time for the current task on a given machine.

TheEarliestMachine determines whether the current task can be processed by the machine with the lowest current makespan among the available machines. In other words, it verifies whether the machine being considered is the most efficient one for the task. If it is not, the function skips it until the most suitable machine is identified.

Time complexity

We have multiple machines, and the process begins by selecting a machine and iterating through all remaining jobs. When a suitable job is found, it is assigned to the machine, and the process returns after assigning one stage of a job. It may be necessary to account for idle time, and in certain cases, the algorithm may iterate through the jobs multiple times. Consequently, the algorithm's upper bound is O (number of machines $\times$ number of jobs $\times$ number of jobs $\times$ number of stages).

Problem 5

To conduct the experiment, we construct 8 scenarios. For each scenario, we generate 50 instances. The descriptions of the setting of each scenario are as follows:

Scenarios	Descriptions
Scenario 1	Let the number of jobs be fixed at 30 and the number of machines be fixed at 5. Each job has 0.2 of probability to have only one stage and 0.8 to have two stages. For each piece of each job, the processing time is uniformly drawn from [0.5, 3.0). For each job, the first piece can be processed by random machines with equal probability, and at least one machine could process it, and similarly, the second piece can be randomly processed by one or multiple machines. For each job, the due time is randomly assigned to be 2, 3, 4, or 5 with equal probability.
Scenario 2	For each job, there is only one machine could process the first piece of job with 0.5 probability. All the other settings are identical to scenario 1.
Scenario 3	For each job, if the job has two stages, it cannot be done by a single machine. In other words, those jobs must be split and completed in different machines.
Scenario 4	For each piece of each job, the processing time is uniformly drawn from [10, 50). For each job, the due time is randomly assigned to be 50 or 90 with equal probability. All the other settings are identical to scenario 1.
Scenario 5	Let the number of jobs be fixed at 25 and the number of machines be fixed at 5. Each job has 0.3 of probability to have only one stage and 0.7 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.
Scenario 6	All jobs have two stages. All the other settings are identical to scenario 5.
Scenario 7	For each piece of each job, each machine is allowed to process it with probability 0.5. All the other settings are identical to scenario 5.
Scenario 8	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 5.

Our basic algorithm schedules jobs by their ID. If both stages of a job can be processed by the same machine, they will be scheduled on that machine. Otherwise, the algorithm checks other machines, considering conditions such as gaps of less than 1 hour between stage 1 and stage 2, and whether a machine must wait for stage 1 to be completed. The algorithm follows a straightforward rule based on job ID and does not consider optimizing the use of specific machines. The following two pages are pseudocodes for simple heuristic algorithm, data, and charts related to our performing results.

```
Algorithm 1 Job Scheduling Algorithm
Require: Import pandas as pd
Define Class Job include (id, s1, s2, m1, m2, due)
tardiness ← 0
df ← pd.read_csv(filepath)
for i in range(len(df)) do
    myJob ← Job(df.iloc[i])
    machines.UpdateSet(myJob.m1,myJob.m2)
    jobs.append(myJob)
end for
for i in range(len(machine_use)) do
    machines_time.append(0)
end for
while len(jobs) ≠ 0 do
    for i in range(len(machines)) do
        if len(jobs) == 0 then
            break
        end if
        if i + 1 in jobs[0].m1 then
            if jobs[0].s2 ≠ 0 and i + 1 in jobs[0].m2 then
                machines_time[i] ← machines_time[i] + jobs[0].s1 + jobs[0].s2
                tardiness ← max(machines_time[i] - jobs[0].due, 0)
            else if jobs[0].s2 ≠ 0 then
                for j in range(len(machine)) do
                    if j + 1 in jobs[0].m2 then
                        if machines_time[j] ≥ machines_time[i] + jobs[0].s1 + 1 then
                            machines_time[i] ← machines_time[j] - 1
                            machines_time[j] ← jobs[0].s2
                        else if machines_time[j] ≤ machines_time[i] + jobs[0].s1 then
                            machines_time[i] ← machines_time[i] + jobs[0].s1
                            machines_time[j] ← machines_time[i] + jobs[0].s2
                        else
                            machines_time[i] ← machines_time[i] + jobs[0].s1
                            machines_time[j] ← machines_time[j] + jobs[0].s2
                        end if
                        tardiness ← max(machines_time[j] - jobs[0].due, 0)
                        break
                    end if
                end for
            else
                machines_time[i] ← machines_time[i] + jobs[0].s1
                tardiness ← max(machines_time[i] - jobs[0].due, 0)
            end if
            jobs.pop(0)
        end if
    end for
end while
Output (tardiness, max(machines_time))
```

Figure 10: Simple Heuristic Pseudocodes

		Simple Heuristic		LR		Heuristic	
Scenario	Metric	Mean	Std Dev	Mean	Std Dev	Mean	Std Dev
1	Makespan	27.05	4.34	5.37	0.36	22.77	2.37
2		31.84	5.58	5.43	0.27	27.88	3.10
3		32.91	6.96	5.42	0.31	26.7	2.79
4		516.61	90.27	90.86	4.74	477.91	60.51
5		59.92	6.16	17.48	1.35	67.7	11.69
6		68.57	6.90	17.85	1.12	91.45	18.14
7		72.81	16.73	17.19	1.16	61.13	9.01
8		59.02	7.82	17.35	1.42	68.12	14.95
1	Tardiness	6066.34	3580.57	14.78	4.85	265.24	34.33
2		19503.51	4397.35	18.96	4.61	338.86	42.32
3		33670.69	4070.91	18.37	5.30	352.04	54.79
4		144665.38	61016.70	195.76	54.08	5767.03	847.83
5		260243.83	7031.84	77.46	15.85	732.79	105.21
6		287352.36	8508.95	101.74	19.85	1032.12	205.51
7		316202.45	8203.17	77.89	18.57	635.82	105.04
8		384937.98	7194.13	100.11	18.55	772.58	146.42

Table 1: Means and Standard Deviations for Makespan and Tardiness

	Makespan		Tardiness	
Scenario	Simple Heuristic	Heuristic	Simple Heuristic	Heuristic
1	4.0349	3.2396	409.3930	16.9434
2	4.8601	4.1324	1027.6374	16.8720
3	5.0709	3.9248	1831.8844	18.1634
4	4.6859	4.2600	737.9978	28.4599
5	2.4282	2.8733	3358.7719	8.4604
6	2.8418	4.1235	2823.3534	9.1446
7	3.2360	2.5565	4058.8279	7.1635
8	2.4014	2.9262	3843.9694	6.7169

Table 2: Optimality Gaps for Makespan and Tardiness

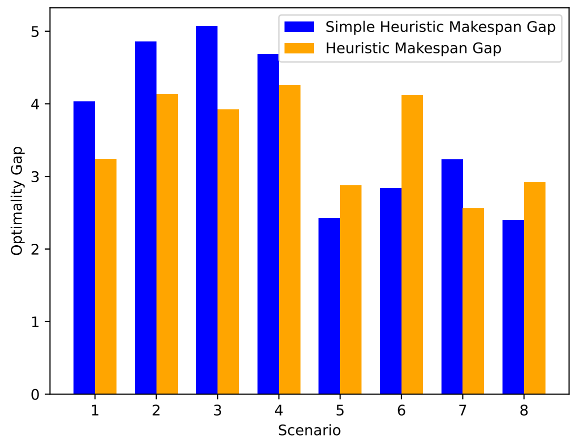

Figure 11: Comparison of Makespan Optimality Gap

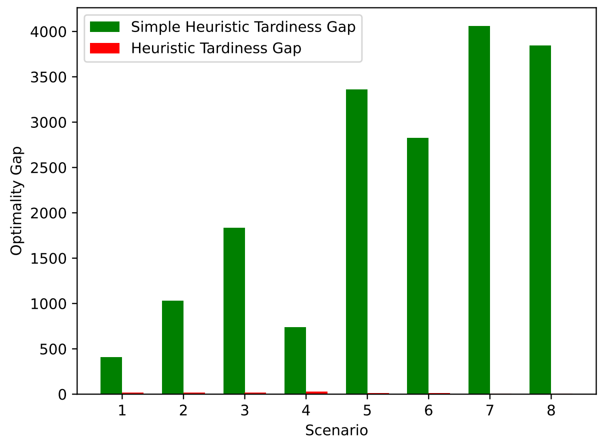

Figure 12: Comparison of Tardiness Optimality Gap

According to Figure 11, in scenarios 5, 6, and 8, the simple heuristic algorithm results in a shorter makespan because, in these scenarios, many of the available machines for each job's Stages 1 and 2 are the same, which favors the simple heuristic algorithm in minimizing the makespan. This algorithm strives to complete each job on the same machine whenever possible, whereas the heuristic algorithm does not have this specific design. Conversely, in scenarios 1, 2, 3, 4, and 7, the heuristic algorithm performs better in minimizing makespan compared to the simple heuristic algorithm. This is because, in scenarios 1, 2, 3, and 4, each job at each stage may use a random selection of one or multiple machines, making it unlikely that Stage 1 and Stage 2 of a job will be completed on the same machine. Similarly, in scenario 7, each machine has a 0.5 probability of processing each job, which also reduces the likelihood of completing Stage 1 and Stage 2 on the same machine. In summary, when Stages 1 and 2 of a job can be processed on the same machine, it benefits the simple heuristic algorithm. Therefore, in scenarios 5, 6, and 8, the heuristic algorithm results in a larger makespan compared to the simple heuristic algorithm, while in scenarios 1, 2, 3, 4, and 7, it performs better in minimizing makespan.

According to Figure 12, in all scenarios, the heuristic algorithm significantly outperforms the simple heuristic algorithm in minimizing tardiness. This is because the heuristic algorithm prioritizes more urgent jobs and allocates the fewest possible jobs to the machines with the least availability, thereby reducing the burden on machines that can handle multiple types of jobs. In contrast, the simple heuristic algorithm processes jobs in ascending order based on job ID, resulting in poorer performance in reducing tardiness.