Operations Research, Spring 2022 (110-2) Case2

楊晴雯 b06102020
張庭瑜 b07602024
王彥普 b08303141
何陞宜 b09705043

7 May, 2022

1 Problem 1: skipped

See the program attached. Below is the results of our heuristic algorithm to TA's 5 instances.

Instance	Tardy	Makespan
1	1	6.9
2	1	8.6
3	3	10.3
4	5	22
5	5	28.7

Table 1: Heuristic results for TA's instances

2 Problem 2: Heuristic Algorithm: LST ratio with find-hole

Our algorithm leverages the ratio of processing time and latest starting time (LST) on job ordering, and tries to re-use the idle periods on machines (i.e. find holes).

1. Defining variables and function

Let

J be the job pool,

O be stages (operations) $\{1, 2\}$,

M be the machine pool,

$J.due\_dates_j$ be deadline $\forall$ jobj $\in J$,

$J.total\_processing\_times_j$ be the total processing times $\forall job$j $\in J$,

$M.fintime_m$ be the finishing time $\forall$ $machine$m $\in M$,

$J.completion\_times$ be the completion times $\forall$ jobj $\in J$.

Define

LST_j to be $due\_date - total\_job\_processing\_time$ $\forall$ jobj $\in J$,

$LST\_ratio_{j,o}$ to be $\frac{stage\_processing\_time_o}{LST_j}$ $\forall job$j$stage$o $\in J$,

$versatility_m$ to be the number of operations machine m is capable of executing $\forall$m $\in M$.

2. Algorithm Steps

1. We call the first stage of a job its *first operation*, second stage *second operation*. For every operation, calculate its associated LST_ratio.
   - The less LST is, the more urgent this operation is.
   - The larger $stage\_processing\_time$ is, the earlier this operation should be started.
   - Therefore, those with **high** LST_ratio should be scheduled first.
2. Jobs are put into a priority queue Job_Q ordered by LST_ratio.
3. Machines are sorted into an array $Mach\_Q$ by $versatility$.
4. `Extract_min()` from Job_Q the next operation to be performed, denote $curr\_op$. In implementation, we use a min queue, so the LST_ratio are multiplied by -1 to that we extract the one with largest LST_ratio. And then it goes through the following procedure:
   (a) We check if $curr\_op$ is a *null* operation, that is, if it is some job's 2^{nd} stage and its processing_time equals to 0; if yes, we schedule it onto machine $None$ and continue with the next operation extracted from Job_Q. Otherwise we go to step 4 (b).
   (b) That is, in order to preserve job operation precedence, sometimes operation 2 must wait until operation 1 completes, which creates idle periods with various length in machine's working schedule. To make use of these idle times, denoted by $hole(s)$, a `find_hole()` method (Algorithm 1) scans through the machines for an available hole. For the function to work, a list of holes $m.holes$ is stored $\forall$m $\in M$.

(c) If `find_hole()` successfully finds a hole h on a machine m', schedule *curr_op* into h and update associated values, including $m'.holes$. If h is exactly the size of *curr_op*'s processing time, it is popped from $m'.holes$; otherwise, h is updated to a smaller size. Otherwise if `find_hole()` fails, it either means that there are no holes on available machines, or that no legal holes are big enough to fit in *curr_op*.

5. If `find_hole()` succeeds, go to step 7, otherwise we try scheduling *curr_op* at the end of a machine's completion time. We calculate the best machine to schedule *curr_op* by getting the subset of machines M' that can execute *curr_op* , and then ordering them by (1) current finished times and (2) their *versatility*. To sum up, the best machine is got by: `min(avail_machines_idx, key = lambda m: (M.fintime[m], M.versatile[m], m))`

6. Then before directly putting *curr_op* onto this machine, We examine if it is a good time to schedule *curr_op* by 2 conditions:

   (a) We compute a temporary completion time, that is the scheduled machine's finished time plus *curr_op*'s processing time. If *curr_op* is found to be already tardy, then it doesn't really matter where we schedule it, and scheduling it now may block other operations and make them tardy as well. Therefore we associate *curr_op* with infinite *LST_ratio*.

   (b) If *curr_op* is operation 2 of some job, its scheduling requires an idle time. We define a value *tolerance* by, where *tolerance_ratio* is a hyper-parameter $\in [0, 1]$:

   $$best\_makespan = \frac{\sum_{j \in J} processing\_time\_j, 1 + processing\_time\_j, 2}{|M|}$$

   $$tolerance = best\_makespan * tolerance\_ratio$$

   if the current idle is too large, i.e., larger than *tolerance*, we postpone it temporarily by associating *curr_op* with $LST\_ratio + K$, where $K \geq 0$ is another hyper-parameter of delayness. In implementaion, we set*tolerance_ratio* to be 0.3 and K as 3.

   (c) If any the above conditions is satisfied, *curr_op*'s $failure$ is incremented to record how many times the job is popped from Q but fails to be scheduled. If $failure$ for *curr_op* $> failure\_tolerance$, we schedule it no matter what. $failure\_tolerance$ is yet another hyper-parameter, which we set to 2.

   (d) If *curr_op* is successfully scheduled, we update associated values, otherwise, and push the updated *curr_op* with new *LST_ratio* back to *Job_Q*.

7. Check if all job operations are scheduled, if yes, stop the algorithm, if no, continue the iteration by going back to step 4.

Algorithm 1 find_hole()

Suppose *curr_op* is (j, o) with processing time p_{ij}
for m in $Mach\_Q$ **do**
    if curr_op cannot be done on machine m **then**
        continue
    end if
    if $m.holes$ is empty **then**
        continue
    end if
    for *hole_start*, *hole_end* in $m.holes$) **do**
        hole_length = *hole_end* - *hole_start*
        ▷ compute *legal_length*: if *curr_op* is first operation, *legal_length* = *hole_length*; otherwise *legal_length* is *hole_end* - operation 1's completion time.
        if *legal_length* $< p_{jo}$ **then**
            continue
        else
            return *hole_start*, *hole_end*
            ▷ a proper hole is found, in implementation we return hole's *id*
        end if
    end for
end for

3. Algorithm Time Complexity

Let n denote number of jobs, m denote number of machines.Since m is not immediately accessible in instance.csv, we need to go through all available machines in all operations for all jobs to get the maximum machine index as m, which takes at most $O(2n \times m) = O(nm)$.

Step 1 takes $O(n)$ for subtraction and division for n jobs. Step 2 heapifies the jobs by LST_ratio in queue in $O(n)$. Step 3 orders the machine by $versatility$, which has the same time complexity as finding machine number $O(nm)$ to compute $versatility$, and $O(m)$ to heapify.

Then for step 4, we enter the iteration of scheduling each of the jobs in job pool J: Step 4 extracts minimum from Job_Q, which takes $O(\log n)$ to keep the heap invariant. 4(b) implements find-hole method: since only n operations (the second operations of the n jobs) could create holes, there are at most n holes across all machines, so that one `find-hole()` takes $O(n)$ for scanning through all holes and doing constant-time condition checking. However, the holes are accumulated while scheduling jobs; in worst cases, each 2^{nd} op creates 1 hole, so we

have the total holes accumulated over iteration as $1, 2, 3, 4, 5, \ldots$. With amortized concept, each $find\_hole()$ actually just costs a slightly better $O(\frac{\sum_{i=1}^{n} i}{n}) = O(\frac{n}{2})$.

Step 5 finds the best machine by choosing the one with minimum finishing time and then minimum versatility, which takes $O(m)$. Then checking the 2 conditions in step 6 can be done in constant time with list indexing; updating associated values also takes constant time.

In conclusion, step 4 iterates through all jobs in $O(n)$ and dominates the runtime. In each iteration we have to spend $O(n)$ for $find\_hole()$ and (if $find\_hole()$ fails) $O(m)$ for finding the best machine. Overall the proposed *LST ratio with findhole* heuristic algorithm takes $O(n(n+m))$ to execute.

3 Problem 3: Heuristic Benchmarking

Heuristic Algorithm Performance

In order to determine the efficiency of our algorithm, we carried out some experiments and did the benchmarking. We went through two stages; in the first stage the 5 TA instances are tested, and in the second self-generated 334 testcases (187 easy, 147 medium, whose difference would be specified in the last section of Problem 3) are tested. In the first stage, our heuristic algorithm **LST ratio with find hole** are compared to 3 benchmarks: 2 algorithms FCFS (First-Come-First-Serve) and Random scheduling, and a time-limited Gurobi IP formulation. The IP is designed so that it has 60 seconds on 1^{st} phase optimizing toward minimum tardiness, and 180 seconds on 2^{nd} phase optimizing toward minimum makespan. Note that our FCFS and Random algorithms implement the step 5 provided in our heuristic algorithm steps above, i.e. **finding the best machine** step, and only job ordering is Random or FCFS, which makes the 2 benchmarks competitive enough. Apart from the 5 TA instances, we picked 2 specific instances from our self-generated testcases to further illustrate the advantage and disadvantage of these algorithms, IP and our **LST ratio with find hole** in particular. The 2 specific instances are denoted as instance 6 (ours performs well) and 7 (ours performs poor) below.

Table 3 lists the results from the 4 algorithms and their performance gap of tardy jobs with respect to time-limited IP. We denote our **LST ratio with find hole** as **LSTratio+FH**, and **time-limited IP** as **TL-IP**. It is clear that LSTratio+FH outperforms the 2 benchmarks in terms of TA's instances. With LST_ratio as job ordering and the method to schedule jobs into holes, we could save a lot more jobs from being tardy than Random and FCFS (as the performance gap is around 60% and 70%). While for time-limited IP, it continues to look good until the last testcase: it fails to produce a feasible solution for testcase 7 within given time, hence the statistics is not recorded. As testcase 7 involves only 36 jobs and 11 machines, which

does not seem to be a large case, we can conclude that IP is not practical in solving real-world scale problems of this sort. On the other hand, our LSTratio+FH takes less than 0.01 seconds to produce a feasible and acceptable schedule, which in turn proves that a heuristic is definitely needed.

As for the second objective makespan, the peformance gaps are around 2% to 10% for the 2 simple benchmarks (the 2 benchmarks outperform our method). However, considering their tardy jobs are already different, it does not seem to make much sense to compare makespans under this circumstance, so we skip the discussion of makespans here.

	testcase		Objective Value (tardy jobs)				Performance Gap		
	job no.	machine no.	TL-IP	LSTratio+FH	FCFS	Random	LSTratio+FH	FCFS	Random
1	12	5	0	1	2	3	-	-	-
2	12	5	0	1	5	5	-	-	-
3	10	5	1	3	4	5	200%	300%	400%
4	15	7	2	5	11	8	150%	450%	300%
5	20	9	2	5	10	11	150%	400%	450%
6	13	9	1	1	4	5	0%	300%	400%
7	36	11	-	26	32	31	-	-	-

Table 2: Comparison of tardy jobs of three methods

testcase	Objective Value(tardy job)			Performance gap w.r.t ours	
	LSTratio+FH	Random	FCFS	Random	FCFS
Easy	12.82	21.11	21.18	65%	65%
Medium	10.92	18.71	18.80	71%	72%

Table 3: Comparison of tardy jobs of three methods

It is worth mention that when running time-limited IP, a great portion of testcases (roughly 20%) either fail to produce feasible solutions or fail to converge to acceptable objective values. For example, in one generated testcase (61^{st} generated instance), with 61 jobs and 17 machines, **IP can only produce a schedule with 60 tardy jobs and 95.8 makespan, while our heuristic can reach a schedule of 47 tardy jobs and 45.6 makespan**. Therefore, in many scenarios heuristic can actually outperform timelimited-IP within less than 1% of the 5 minute time that the latter is given. It again fortifies our previously-established argument that heuristic is a must in solving practical, over-medium-scale problems.

Testcases generation setting

We generated 2 types of testcases, **easy** and **medium** and manually removed those with weird formats or not conforming to our conditions, and finally tested the 4 algorithms with 187 easy and 147 medium testcases.To ensure that the testcases are with good quality, we made 4 rules (or constraints):

1. **Machine constraint**:
   For **easy**, all jobs' 1^{st} operation can be scheduled on all machine in M, and for one testcase, we run only 1 random picking a subset of machines M' out of M, and all 2^{nd} operations of the jobs within the same testcase are associated with M' as its available processing machines. For **medium**, all stages' processing machines are randomly picked, so we run random picking $2n$ times for each testcase with random seed 2022.

2. **Due time calculation rule**:

   $$due\_time = (processing\_time\_1 + processing\_time\_2) \times 1.2 + ND\_error \times 5 + 2$$

   This rule applies to all jobs in both **easy** and **medium** cases. ND_error is the random term picked from normal distribution: it has 68% probability to return a value within range $(-1, 1)$; 95% to return a value within $(-2, 2)$, and so on. This is to ensure that processing times and due times have a positive relationship: the more difficult a job is, the later its deadline should be.

3. **Total processing time constraint**:

   $$total\_processing\_time \leq |M| \times 72$$

   where M is the set of machines. This is to ensure that in average each machine does not work more than 3 days, which is our perception of a reasonable workload limit that a machine should be scheduled in one day. This rule applies to all jobs in both **easy** and **medium** cases.

4. **Total must-tardy ratio constraint**:
   Since due time calculation rule involves an random error term, it is likely that $due\_time - (processing\_time\_1 + processing\_time\_2) \leq 0$, which makes the job definitely tardy. If cases like these overflow, then the testcase fail to evaluate the performance between algorithms. Therefore, we constrain the number of these jobs down to 20% of the job set J. This rule applies to all jobs in both **easy** and **medium** cases.

4 Appendix

Heuristic Gantt Chart

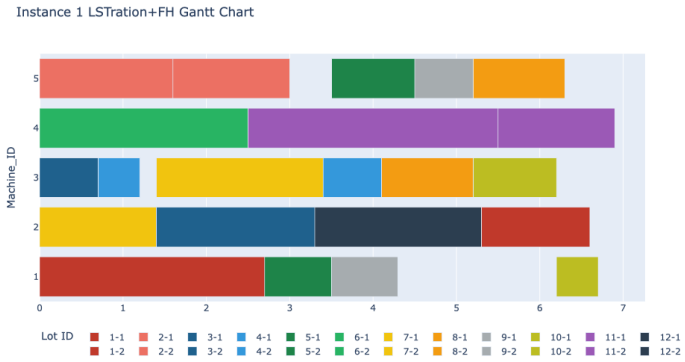

Figure 1: Heuristic for instance 1

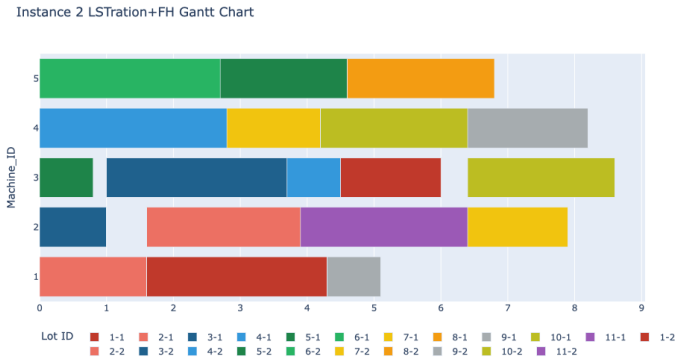

Figure 2: Heuristic for instance 2

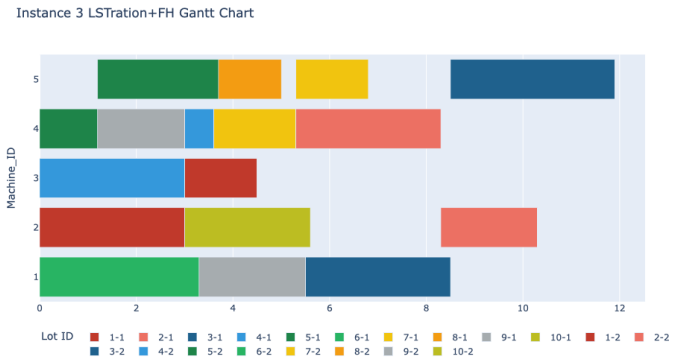

Figure 3: Heuristic for instance 3

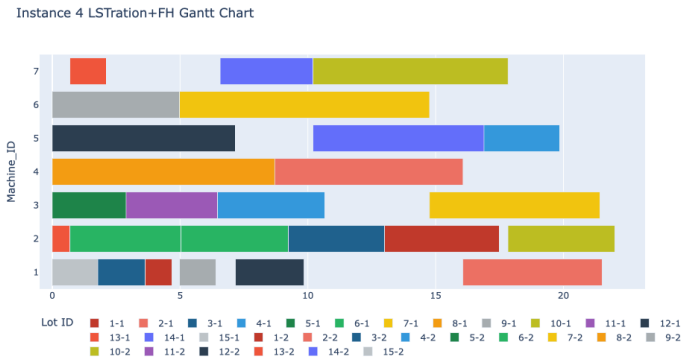

Figure 4: Heuristic for instance 4

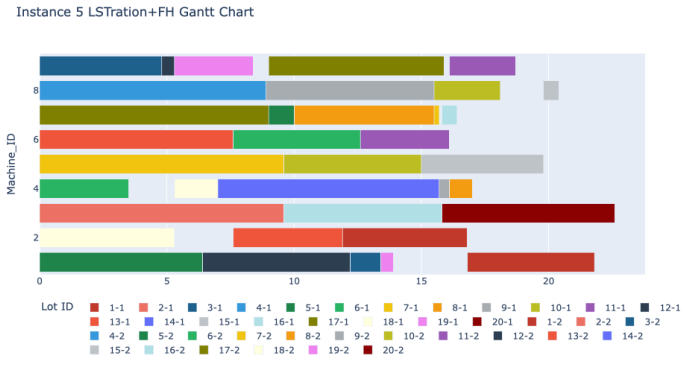

Figure 5: Heuristic for instance 5

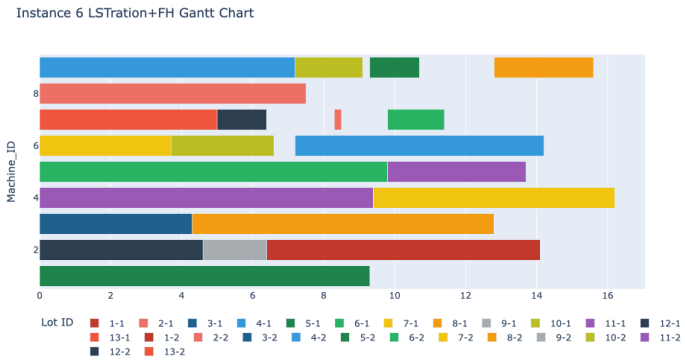

Figure 6: Heuristic for instance 6

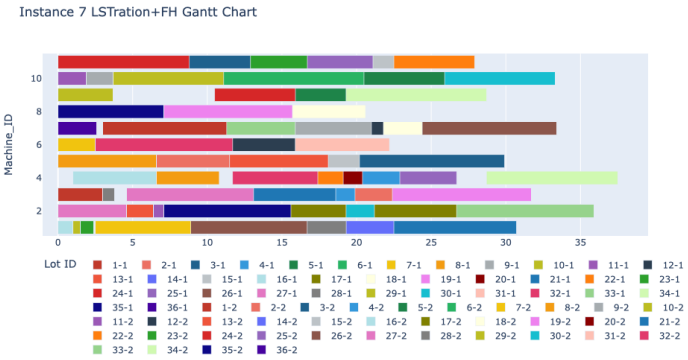

Figure 7: Heuristic for instance 7