Simple Heuristics for Scheduling Apache Airflow: A Case Study at PT. X

Hans Tanto,[1, a)] Indriati N. Bisono,[1, b)] and Hanijanto Soewandi[1, 2, c)]

[1]*Universitas Kristen Petra, Surabaya 60236, Indonesia*
[2]*MicroStrategy, Inc., VA 22182, USA*

[a)] hanstanto13@gmail.com
[b)] Corresponding author: mlindri@petra.ac.id
[c)] hsoewandi@microstrategy.com

Abstract. In the big data era, there will be a lot of data that needs to be handled properly for making well-informed business decisions. As a result, the data needs to get updated regularly for relevant analysis. Apache Airflow is commonly utilized to schedule the data update by running the query through sequences of DAG tasks. Unfortunately, scheduling DAG tasks is categorized as NP-Hard problem which is difficult to obtain the global optimum solution. This paper shows a simple heuristic algorithm to solve the subset of DAG tasks at PT.XYZ using 2 computers (virtual machines). This is a special case of $P_m|prec|C_{max}$ problem where there are 2 computers are being used. Before constructing the algorithm, CPM (Critical Path Method) is used as the baseline to get the most optimal solution. A Knapsack problem is then used to add additional tasks for virtual machine #1, and a heuristic for virtual machine #2 as a compromise to balance the cost and time of completion. The characteristics of this simple algorithm is discussed with numerical examples/experiments.

Keywords: Heuristics, Apache Airflow, Directed Acyclic Graphs, Critical Path Method.

INTRODUCTION

Apache Airflow is a tool for monitoring, describing, and executing the task based on sequential workflow [1]. Airflow was developed by Airbnb in 2014 to manage complex corporate workflow. Creating an airflow allows users to systemize monitoring process automatically via airflow self-service (user interface). The autonomous airflow system uses Directed Acyclic Graphs (DAGs) to manage the task dependencies and the workflow. DAG run on an interval basis for scheduling or on triggered event.

There are 3 types of air flow used by PT. X, which includes airflow-google cloud platform, airflow machine learning, and airflow-analyst. Airflow-google cloud platform is the domain for data engineers to handle their day to day activities. This includes scheduling ETL data pipelines, pulling data from microservices to data marts, sending data stream jobs, and sending data to third parties from PT. X. Airflow machine learning is typically used by data scientists to perform machine learning data training with new incoming data. On the other hand, airflow-analysis is used to schedule queries to update data marts, and generate reports periodically. In this paper, we focus on using the airflow-analyst to update the data mart on a daily basis and estimate how many machines are needed to support the process using a case study illustration.

First thing to do before the airflow can manage and schedule task is to create a DAG. The airflow-analyst can also set the schedule and intervals to run the DAG. Moreover, it can notify the owner that the process has failed after several trials. The number of trials may vary from one to another depending on the user's settings. DAG creation is simply several tasks that must be done within specified dependencies among the tasks. The simplest DAG involves sensor table tasks, delete rows, and load data by appending task.

Generally, there are several task types that exist in airflow scheduler:

1. Big Query load operator, to load data to big query using query that the user previously used in creating a data mart.
2. Big Query delete row operator, to delete rows from table using query to prevent the duplicate records.
3. Big Query table sensor operator, to check the table availability before running the sub sequential process.
4. Big Query update table metadata operator, to get the schema of the related data mart.
5. External task sensor operator, to sensor another DAG. It will be beneficial when a particular DAG can be run after other DAG running process finishes.
6. Dummy operator does nothing, it is used to make DAG easier to read.

LITERATURE REVIEW

In dealing with big data, there are a lot of data marts that needs to be updated regularly. The challenge is to update the data marts in a short time with minimum exerted resources. Furthermore, there's also a high chance that each data mart is dependent on the other data marts. A delay in updating one data mart which is caused by manual work will affect the other dependent data marts to fail in updating. DAG task is the solution to overcome this problem since it specifies the dependencies and make sure that the data is scheduled to update optimally and automatically.

Scheduling with DAG precedence in Apache Airflow is similar to a very well-known subject in Project Planning and Scheduling. Following Pinedo[2], if there are unlimited virtual machines that can be used, the problem is essentially the famous Critical Path Method (or Project Evaluation and Review Technique) in Project Planning that has been discussed in various Operations Research literature, *e.g.*, see Hillier and Liberman[3] (2021) – denoted as $P_\infty|prec|C_{max}$ problem. On the other hand, if there is only one computer to use, the problem becomes $1|r_j, prec|C_{max}$ in which Lawler's algorithm, *i.e.*, dynamic programming – Lawler[4] is known to produce an optimal solution in polynomial time.

However, when the number of computers (or machines) are finite, then the problem is known to be an NP-Hard where the global optimal is difficult to obtain. With a finite number of computers, Apache Airflow scheduling can be denoted as $P_m|prec|C_{max}$ problem, *i.e.*, the problem of minimizing makespan on parallel machines with precedence constraint. Pinedo[2] (Section 5.1) discussed various heuristics for this type of problem as well as special cases of the precedence when an optimal solution can be obtained. Recently Hua *et al.*[5] proposed a deep learning algorithm to improve the heuristic solution for this NP-Hard problem.

In our case, we simply developed heuristic algorithms for this case study based on the sample problem as shown in Fig. 1. Before we discuss further our heuristic algorithms and how we take advantage of the characteristics of our Apache Airflow problem, we will explain a bit about the use case of Airflow scheduler at PT. X.

AIRFLOW SCHEDULER IMPLEMENTATION

We had a chance to utilize the airflow analyst to schedule the query (systemize the ETL process). Later on, the scheduled query will automatically add the new data into a targeted data mart. Recalling back that the main data source is prepared by the data engineer team from the microservices OLTP (Online Transactional Processing). OLTP processes transactional activities and focuses more on the customer side [6]. In general, OLTP processes the data at a fast speed since a smaller set of data is used. After the data is processed by OLTP, the data will be sent to OLAP (Online Analytical Processing). OLAP is essentially a data warehouse where a highly relational database is stored [6]. The data warehouse in Big Query is a form of OLAP database that can be used for multi-dimensional reporting and analysis.

The data warehouse in Big Query also needs some pre-processing steps (ETL process) before it can be used directly for the front-end analysis. The reason is because the nature of its size is relatively big since it contains all dimensions from the OLTP. But not all dimensions are important for the front-end analysis, so it will be useless and costly to include them. Therefore, creation of a smaller set of data which is known as a data mart is adopted. This data mart only contains the relevant dimension for the analysis. Thus, the size can be cut down drastically. Furthermore, this data mart is scheduled regularly using an airflow to update the new data coming in. This chapter will illustrate the implementation and step by step process to create a DAG for updating "*Layanan Pendukung"* data mart which is a data source for the business dashboard.

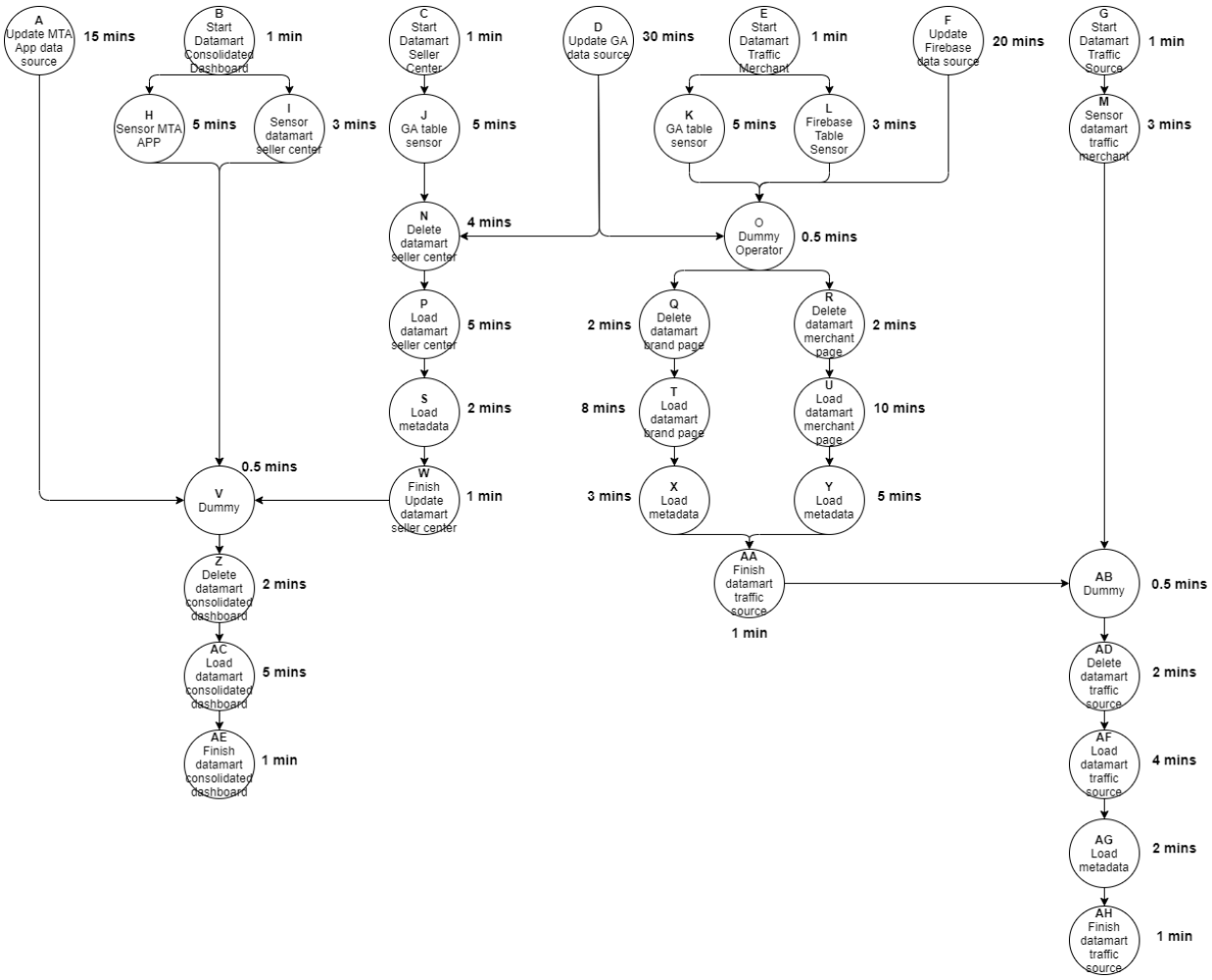

FIGURE 1. Sample Apache Airflow problem

Several steps were accomplished to schedule "*Layanan Pendukung*" data mart using an airflow:

1. Creating a data mart in the Google Big Query. This step must be done before creating DAG in the airflow. Because in the airflow, it requires the user to point on the related data mart to be scheduled.
2. Creating a new DAG by specifying the series of tasks that must be followed by the airflow. The tasks must be set in the right order, otherwise it will not work as it is supposed to be. The result of missed specifying will be the failure of DAG while running and delayed data. For this project, the DAG consists of 6 sequential tasks, as follows.
    a. First task will be checking the source table availability by using a big query table sensor operator. We need to give the tasks name, specifying the task type which is a big query table sensor operator. In addition, we must set the details of the project name in Big Query along with the dataset name and the table name.
    b. Next step is checking the main data source from the OLTP data whether it has been updated on that day or not. For this task, we used a big query table data sensor operator and specified the table details for the respective main data source to be checked. Furthermore, because the first task must be completed before this task, then we need to set the dependencies with the previous task.
    c. After that, a dummy operator is used to separate between the sensor operator to check the data and main tasks to load the data. It will make the DAG more readable for monitoring purposes only.
    d. Following the dummy operator, before loading the new data into the data mart. It must be initiated by deleting the data which occur on the same day as the execution date. The goal is to prevent a duplicate record of the data. Big Query delete row operator is used as the task type. The delete query command used is pointing to the related data mart with filter on the date that matches with the execution date:

DELETE
FROM **`project 1. Dataset name. layanan_pendukung`** -> data mart
WHERE event date = DATE ('{{**execution date**}}', 'Asia/Jakarta')

e. The last one step before loading the data is creating and loading the metadata and getting the schema of the related data mart. Big Query update table metadata operator is used for this condition. The schema will be automatically loaded when the project and table name is correctly specified in the previous delete row operator task. We also carefully set the policy tag for the column that has a personally identifiable information (PII) tag in order to preserve the data security.

f. The last step before turning on the DAG is loading the data into the related data mart which was already created before. Big Query load operator is used to perform this task. We also specified the correct table name which is *layanan pendukung* so the data can be loaded into the right table with the right schema. For load operator, we chose write_append to add the distinct data that do not exist in the data mart. Write_truncate can be utilized as an option, it performs to add the data by overwriting the existing data. The query is modified by changing the source table name with the dynamic table name that can follow the execution date as follows:
from`project-1.dataset_2.ga_sessions {{**execution_date**.in_tz('Asia/Jakarta').strftime('%Y%m%d') }}` ga

3. During the creation of the DAG, we also set the DAG settings. DAG settings contain name, time to schedule, interval for scheduling, the owner identity, number of trials before failure, and e-mail address for notification if DAG task failure occurs.

SIMPLE HEURISTICS FOR APACHE AIRFLOW DAG SCHEDULING

Consider our Apache Airflow DAG scheduling problem that can be depicted as Activity on Node in Fig. 1. Originally, if we assume we can spin off infinite numbers of virtual machines (VMs) to run the DAG tasks, we can finish the problem in 58 minutes (as given by critical path: D → O → R → U→ Y → AA → AB→AD → AF →AG → AH). This represents the best time duration (makespan) that Apache Airflow can finish. As a matter of fact, a simple schedule with 3 machines can finish the Apache Airflow DAG scheduling problem within 58 minutes. Similarly, if we only allow one machine to finish the Apache Airflow DAG, a topological sort will give us a feasible schedule that finishes in 154.5 minutes. Therefore, we focus on analysing the possible solution by using 2 machines only as a compromise to help reduce cost and still finish the Apache Airflow schedule reasonably fast.

Several important characteristics of our Apache Airflow DAG scheduling problem at PT. X that will help us to design our heuristics:

- First, there are not that many parallel successor jobs other than the first 7 jobs that can be started at the beginning. One job has at most 3 successors.
- Second, it has a *characteristic that few of the longer jobs are actually at the beginning of the precedence constraint*. In Fig. 1, the longer jobs are jobs A, D, and F with processing times 15, 30, and 20 minutes respectively.

Therefore, we can try to find a compromise solution to help reducing the cost of the machines and still finishing it around as soon as possible. In this particular case, we can take advantage of the characteristics of the jobs and propose the following heuristic algorithms to minimize the makespan problem by using 2 machines.

First Heuristic Algorithm

The first heuristic algorithm (see Fig. 2) that we built consists the following steps.

1. Calculate Critical Path – assume we can use unlimited numbers of virtual machines (computers) – to find the fastest scheduling (best case scenario). In this example, CP = 58 minutes with schedules: D → O → R → U → Y → AA → AB → AD → AF → AG → AH. Assign this CP schedule to computer VM1.
2. Calculate the total time (*TT*) with topological sort by utilizing only one machine and it produces *TT* = 154.5 minutes (worst case scenario).
3. Calculate $\frac{TT}{Number\ of\ VMs} = \frac{154.5}{2} = 77.25$ minutes. Since the processing times of all jobs are in the multiple of 0.5 minutes, we can round up this number to 77.50 minutes. This will be our target makespan, *i.e.*, the

lower bound of the optimal makespan should be 77.50 minutes if we use 2 Virtual Machines to process the Apache Airflow DAG.

4. Sort the remaining jobs that are not in the CP and do NOT have precedence constraint according to the Longest Processing Time (LPT rule), *i.e.*, {F, A, B, C, E, G} with times {20, 15, 1, 1, 1, 1}. We consider LPT rule because it is known to have worst-case performance bound of 4/3. Furthermore, we choose from the beginning because those sets of jobs tend to have longer duration.
5. Assign jobs in step (4) one by one to VM1 to get a value which is as close as possible to $LB(C_{max})$. Recalling back that the CP value of this case is 58 minutes, cif we add job F to VM1 which handles CP jobs, the *TT* for this machine will be become 78 minutes $(= 58 + 20) \geq LB(C_{max}) = 77.50$. So, we stop. Notice that at this step, we have NOT decided where to put the job F within the schedule in VM1.

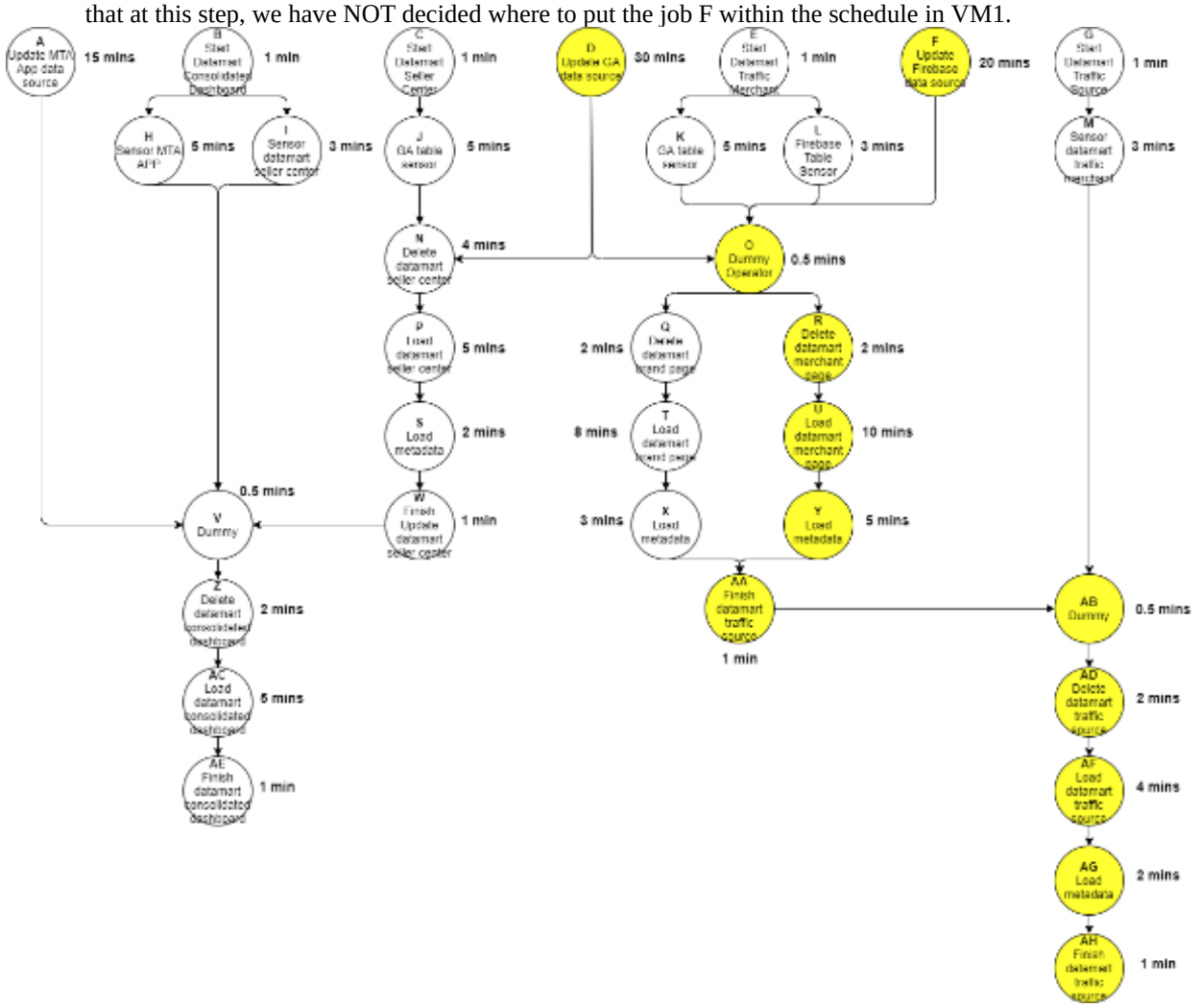

FIGURE 2. The first algorithm problem diagram

In assigning an additional job F to the CP, the algorithm requires to measure the total of its next successors. The job which has ***more successors on the unassigned jobs*** and have no precedence constraint will be assigned earlier compared to those who have less successors. For the first algorithm, since job F has no successor in unassigned jobs, while job D has one successor, then assign job D earlier than F. VM1 will process as following jobs:

VM1: D → F → O → R → U → Y → AA → AB → AD → AF → AG → AH

6. For the remaining jobs on VM2, the problem becomes the famous head-body-tail $1|r_j, d_j|L_{\max}$ scheduling problem which is still an NP-Hard. The release time (r_j) for some jobs (*e.g.*, jobs N & Q) is introduced as the result of scheduled in VM1 above. The precedence for VM2 can be seen on the left hand side of Fig. 2 by eliminating the job that is in VM1 (the yellow circle). However, some jobs (*e.g.*, jobs K, L, M, and X) to be assigned in VM2 are also a precedence for some jobs in VM1 (CP). Therefore, they will *need to have a*

strict due-date (d_j) in order to avoid affecting the critical path in VM1. The release time and due date for the jobs to be sequence on VM2 are given in the Fig. 3.

Jobs (j)	A	B	C	E	G	H	I	J	K	L	M	N	P	Q	S	T	W	X	V	Z	AC	AE
r_j	0	0	0	0	0	0	0	0	0	0	0	30	0	50.5	0	0	0	0	0	0	0	0
d_j				50	68.5				50	50	68.5			67.5		67.5		67.5				
t_j	15	1	1	1	1	5	3	5	5	3	3	4	5	2	2	8	1	3	0.5	2	5	1

FIGURE 3. The release time and due date for the jobs to be sequence on VM2

We create these jobs as standalone sets, and we have the following 3 sets that have release time and strict due-date: {E → K, L}, {Q → T → X}, and {G → M}, *i.e.*, in the above table each set are denoted with different color. For jobs E, K, and L, they need to be finished before the schedule of job O in VM1 started (30 + 20 = 50). Create a set for E, K, and L and specify its latest starting time by 50 – (1 + 5 + 3) = 41 for E as predecessor. The schedule for this set will be {E → K, L} (E starts first because it's a predecessor, while K & L can be assigned freely). The second set, that covers jobs {Q → T → X}, can start any time between [50.5, 67.5 – (2 + 8 + 3) = 54.5] for Q as its predecessor. The third set for this problem includes jobs {G → M} that can start anytime between [0, 68.5 – (1 + 3) = 64.5] for G as its predecessor. The most important thing is that the jobs inside a set must be done sequentially one to the other. For example, once job E finished, we cannot assign other jobs to VM2 since it needs to process job K and job L.

For the rest of the jobs, we apply the Lawler algorithm for $1|r_j, prec|C_{max}$ scheduling problem that works as follows: "Schedule job with the longest processing times as soon as it becomes available according to the precedence constraint." There are multiple sequences on VM2 that can be produced:

- VM2 Option 1: A → B → H → I → C → J → N → P → S → E → K → L → Q → T → X → G → M → W → V → Z → AC → AE
- VM2 Option 2: E → K → L → G → M → A → B → H → I → C → J → N → P → Q → T → X → S → W → V → Z → AC → AE

The result of First Heuristic can be displayed with the following Gantt chart (using option 1 for VM2) and it has C_{max} = 78 minutes, see Fig. 4. Although the first algorithm has a satisfying result, we can improve further for the case of 2 Virtual Machines. It leads to the second algorithm as follows.

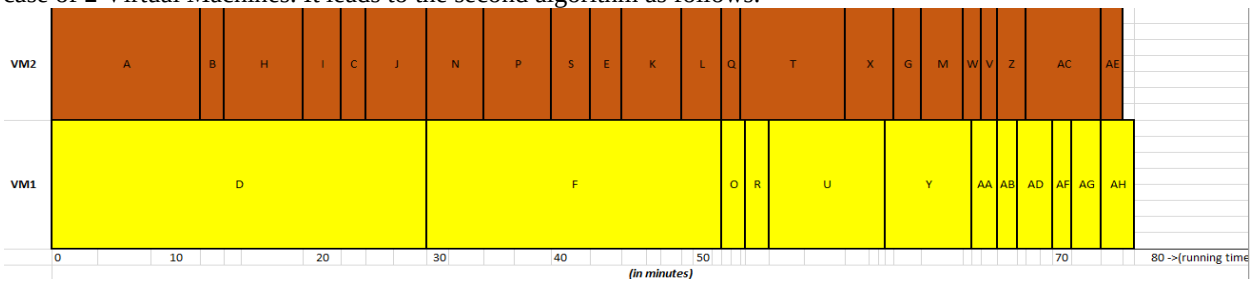

FIGURE 4. Gantt chart for First Heuristic option 1

Second Heuristic Algorithm

In the first algorithm, the *TT* produced (78 minutes) is actually slightly higher than the optimal value. It was caused by assigning job F which drag the *TT* value directly becomes 78. For the second heuristic algorithm, we will do it differently by adding several jobs into the CP to be handled by VM1 by solving a Knapsack problem (see Fig. 5). The first 3 steps (steps 1 – 3 are identical to above).

4. We can replace Step 4 in the First Heuristic algorithm with a better approximation. In the first algorithm, the *TT* is slightly higher than the optimal value. It caused by assigning job F which cause the *TT* value becomes 78. For improvement of this algorithm, we can try to add jobs into the CP which handled by VM1 by solving the following Knapsack problem:

$$\begin{aligned} \text{Max} \quad & 20x_F + 15x_A + x_B + x_C + x_E + x_G \\ \text{s.t.} \quad & 20x_F + 15x_A + x_B + x_C + x_E + x_G \le 77.50 - 58 = 19.50 \\ \text{where:} \quad & x_F, x_A, x_B, x_C, x_E, x_G \in \{0,1\} \end{aligned}$$

Solving the above Knapsack problem produces $x_F = 0, x_A = x_B = x_C = x_E = x_G = 1$. Even though, Knapsack problem is also NP-Complete, there are many efficient algorithms to solve it. Furthermore, the problem is also for much reduced size since we only consider the jobs that do not have precedence.

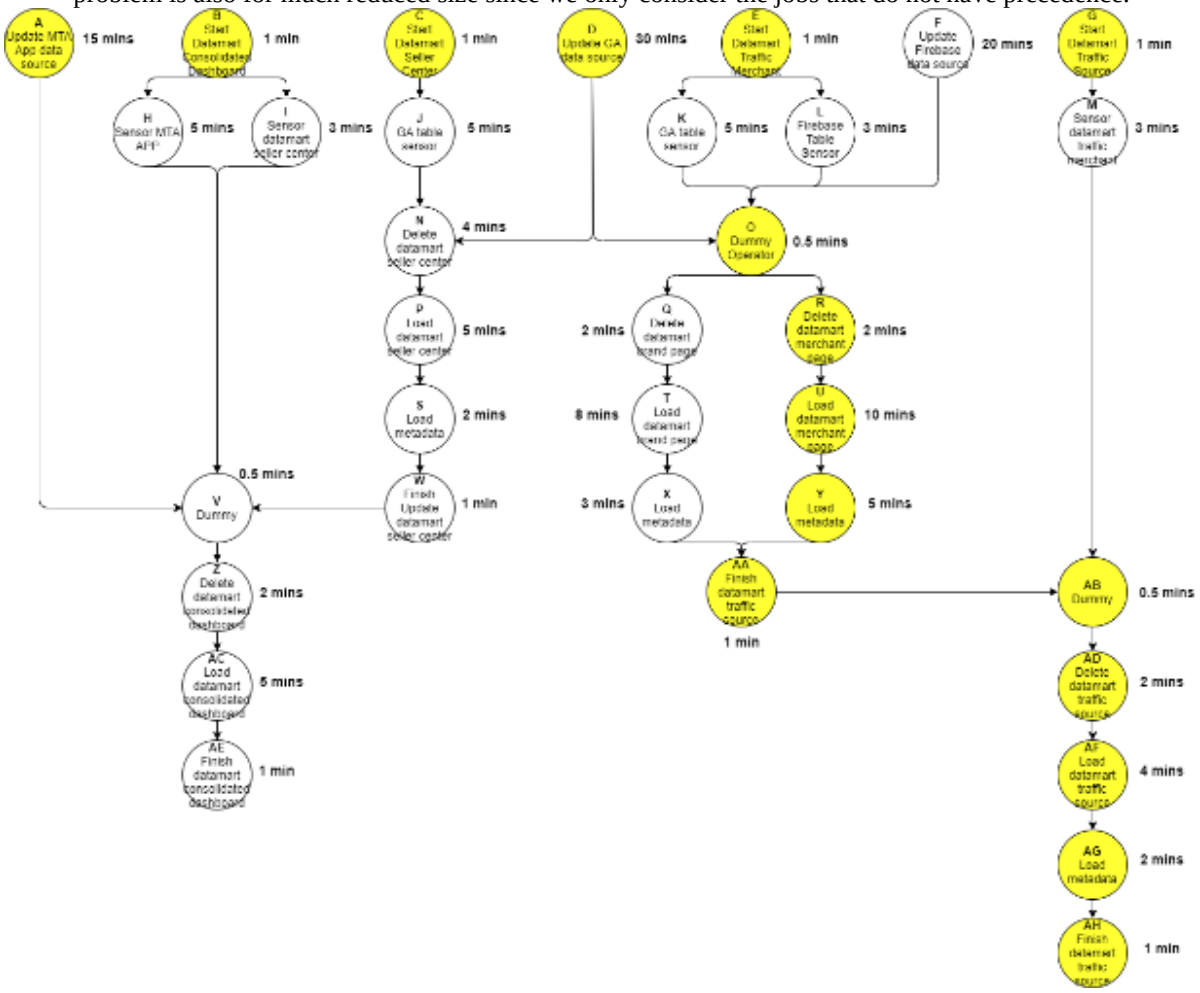

FIGURE 5. The second algorithm problem diagram

5. Assign jobs A, B, C, E, and G to the CP to be handled by VM1 and follow the same principle from the First Heuristic algorithm, the job with more successors in the unassigned jobs and has no precedence constraint will be put in the first order. The total successors for {A, B, C, E, G} on the unassigned jobs are {1, 2, 1, 2, 1} respectively.
   If ties occur (both jobs have the same total unassigned successors), put the job based on the topological sort order of its successors. For this case, both job B and E have the same number of successors and their successors have the same hierarchy level. Thus, job B and E can be assigned freely. However, for job A, C, G, and D which have only one unassigned successor. Job C & G have higher successor topological sort ranking than job A & D, as a result job A will be done after job C & G finished. Thus, the *TT* for VM1 that handles jobs in the set of jobs in CP ∪ {A, B, C, E, G} becomes 77 minutes.
6. Again, for the remaining jobs to be sequenced in VM2, it produces the following tables release time and strict due-date due to the sequence in VM1, as can be seen in Fig. 6.

Jobs (j)	F	H	I	J	K	L	M	N	P	Q	S	T	W	V	X	Z	AC	AE
r_j	0	1	1	3	2	2	4	34	0	49.5	0	0	0	0	0	0	0	0
d_j	49				49	49	67.5			67.5		67.5			67.5			
t_j	20	5	3	5	5	3	3	4	5	2	2	8	1	0.5	3	2	5	1

FIGURE 6. The release time and due date for the jobs to be sequence on VM2 for the second heuristic

Notice that there are 5 sets of jobs that need to be scheduled first, namely: {F}, {K}, {L}, {M}, and {Q → T → X}. Following the same approach as in the First Heuristic algorithm, we can produce the following sequence (again, there are several options for VM2 sequence):

- VM1:B → E → C → G → D → A → O → R → U → Y → AA → AB → AD → AF → AG → AH
- VM2:F → H → J→ K → I → L → M → N → P → Q → T → X → S → W → V → Z →AC → AE

The result of Second Heuristics can be displayed with the following Gantt chart and it has C_{max} = 77.5 minutes

Finally, the second algorithm yields total completion times *TT* for each machine as 77 (for VM1) and 77.50 minutes (for VM2) respectively, thus the *makespan* to finish all the jobs according to this algorithm is 77.50 minutes (the maximum time among machines). The Gantt chart of this schedule is given in Fig.7.

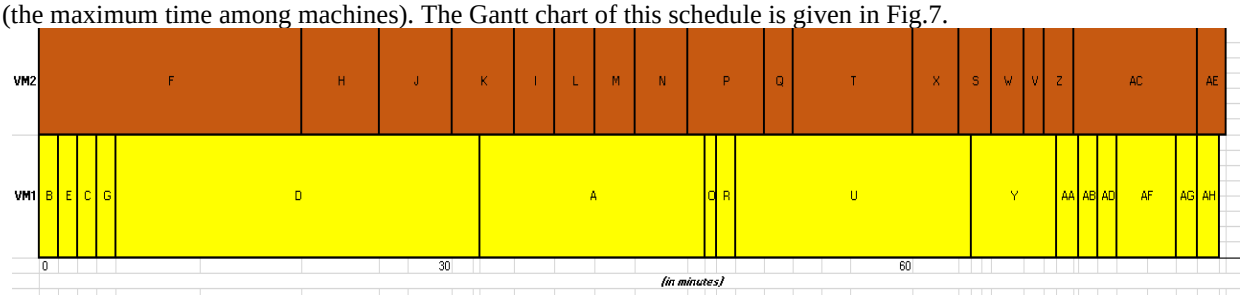

FIGURE 7. Gantt chart for the second heuristic

The second algorithm shows an improvement compared to the first algorithm and hence it's more optimal although the difference is only 0.5 minutes. Furthermore, if we're looking carefully from the problem diagram in Fig.1, in our case, there's no job that has a duration 0.25 minutes. The shortest job takes 0.5 minutes to finish it and hence it is impossible to get a lower value for this problem. Therefore, we can conclude that the second algorithm already reached the optimal value for 2 machines problem.

SOME NUMERICAL EXPERIMENTS

In order to evaluate the robustness of the algorithm, we run a numerical experiment and calculate their deviation. For the first problem, we randomly generated processing times for jobs A, D, and F from uniform distribution with deviation ± 3 minutes and other jobs remain the same processing times (this is a reflection of the real situation that we experience at PT. X). Furthermore, we also tried to extend the usage of our algorithm on more complex real problem as depicted in Fig. 8. Similarly, for this problem we generated processing times for jobs A, C, F, J, and AK to have similar deviation. Subsequently, we established a Confidence Interval (CI) and compared the ratio of C_{max} of our heuristics to the lower bound to see the algorithm performance on each problem.

1. First Problem:

With 95% CI or α = 5%, its C_{max} to finish all jobs is between 76.23 minutes and 81.57 minutes, *i.e.*, 95% $CI = 78.9 \pm 2.67\ minutes$. This deviation occurs due to the randomness on processing time of jobs A, D, & F, as well as the heuristic algorithm scheduling on each trial. The heuristic algorithm performs very well with only 3.38% deviation from its average makespan, and the average worst-case heuristic ratio to its lower bound, i.e., average of $\left(\frac{C_{max}^{H2}}{LB}\right)$ = 1.0038.

2. Second Problem:

Compared to the first problem, the heuristic algorithm performs worse that is indicated by the higher value of its deviation, *i.e.*, 95% $CI = 140.7 \pm 13.20$ minutes (9.38% from its average value). In addition, it has larger average

worst-case ratio compared to its lower bound, *i.e.*, average of $\left(\frac{C_{max}^{H2}}{LB}\right)$ = 1.0995 which is much larger than the first problem.

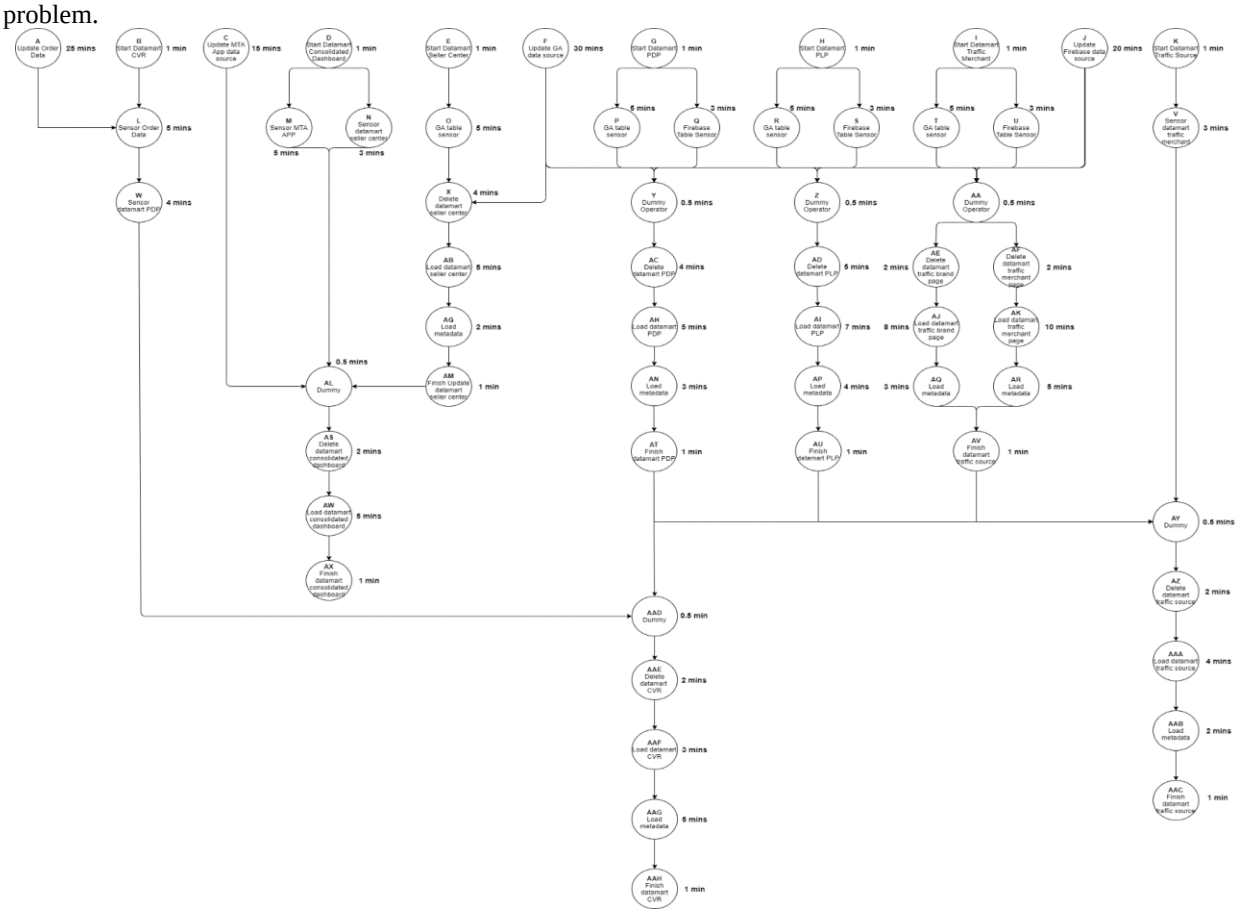

FIGURE 8. An Advanced Problem Diagram

A bit more analysis reveal that the second problem is actually more challenging to solve. If we want to finish within the critical path time, it will need more machines to support the completion of all jobs. Ideally, it requires 5 machines to achieve the most optimal makespan of 58 minutes (CPM path is F → AA → AF → AK → AR → AV → AY → AZ → AAA → AAB → AAC. Finally, since the ideal machine requirement is higher (5 machines) than the first problem (3 machines), thus effort to use only 2 machines tend to have a larger bound.

CONCLUSION

Scheduling DAG tasks on arbitrary machines (> 1) is known as an NP-Hard problem, thus finding a global optimal solution may not be easy and requires many iterations (time). Furthermore, we will most likely never be in the best-case circumstances. In this paper, we discuss our experience working with Apache Airflow at PT. X and outline a new heuristics algorithm to address this discrepancy by taking advantage of some jobs' characteristics in our Apache Arrow DAG tasks used at PT. X (Fig. 1). We use multiple boundary conditions (Critical Path and a single machine) as baseline for formulating our algorithm to arrive at a compromise solution that helps reduce the cost of having virtual machines, but still completes all tasks within a reasonable time.

Our first algorithm can yield a reasonably good solution (78 minutes) which is slightly higher than the target value (77.50 minutes). Refining the algorithm further, our final solution can help schedule DAG tasks to optimally complete with 2 Virtual Machines. Unfortunately, we have not done enough numerical experiments to see the performance of our heuristics. This should be the next line in our research.

Although this algorithm provides its efficacy for solving the problems stated in this paper, there are also some limitations. First, in building the algorithm, we do not consider the scheduled time to start the DAG task. Instead, we assume that all DAG tasks can be started at any time following a scheduling order as long as it has no precedence

constraint. Second, the problem we formulated is to use a constant duration time which is least likely to occur in reality. Future research could incorporate different release time for jobs, as well as PERT techniques to address variations in job duration.

REFERENCES

1. Wikimedia Foundation, Apache Airflow, Wikipedia. (Retrieved at March 16, 2021).
2. M. L. Pinedo, Scheduling: Theory, Algorithms, and Systems, 3rd ed (Springer, 2008), pp. 115-120.
3. F. S. Hillier and G. J. Lieberman, Introduction to Operations Research, 11th ed (McGraw-Hill, 2021), pp. 399.
4. E. L. Lawler, Optimal Sequencing of a Single Machine Subject to Precedence Constraints, Management Science 19, 544–546 (1973).
5. Z. Hua, F. Qi, G. Liu, and S. Yang, Learning to Schedule DAG Tasks (2021), pp. 1 – 13. Retrieved from: https://www.researchgate.net/publication/349880364_Learning_to_Schedule_DAG_Tasks
6. G.S. Reddy, R. Srinivasu, M. P. C. Rao, and S. R. Rikkula, Data Warehousing, Data Mining, OLAP and OLTP Technologies are Essential Elements to Support Decision Making Process in Industries, International Journal on Computer Science and Engineering 2, 2865–2873 (2010).