Job shop operator scheduling software with change-over times

Holden Milne & Dr. Opeyemi Adesina (Supervisor)

Computer Information Systems - University of the Fraser Valley

Abstract

We present a solution to a variation of the Job Shop scheduling problem in which jobs (with each job having assigned priorities) are to be run by m machines and serviced by n operators. In a job shop, different machines run concurrently, which may result in multiple jobs being completed at the same time. It becomes a concern whenever jobs with lower priorities are serviced over jobs with higher priorities. Our goal is to develop a software to guide operators on jobs to be serviced at any given time. We have formulated several algorithms based on the knapsack and leader election problems to realize our goal each with $O(m^2)$ run-time complexity, where m is the number of machines. But we limit our discussion here to the np-hard knapsack problem.

Background

The issue we address considers a situation where at least two machines complete their cycles at almost the same time. An operator may spend time changing over a machine with lower priority while a machine with greater priority is left idle. In a job shop, time is money, therefore minimizing the service down time for high priority jobs becomes critical.

This problem is similar to a well studied problem known as the Job Shop Problem. However, in this variant we are scheduling when the machines will be changed over by an operator. A key difference is that we have to account for the machines change-over time based on some user-defined priorities.

Here we will discuss a solution formed around the Knapsack problem. The Knapsack Problem is where given a knapsack with capacity C and a list of objects with known weights and values we want to find the optimal total value in the knapsack while keeping the total weight less than C. If the list of objects is infinite, the Knapsack Problem is unsolvable in a reasonable amount of time. If it is finite, however, it has many different solutions.

Three important parameters about the machines we will be working with are.

1. Change-Over Time (COT) – The time it takes for an operator to make an idle machine runnable again.
2. Time to Completion (TTC) – The time remaining for the machine to finish its run cycle.
3. Priority (P) – The importance of the job the machine is currently executing.

Problem Formulation

Our goal is to determine a change-over schedule where higher priority machines tend to be closer to the front of the schedule while ensuring that every machine will have a fair chance to get serviced. Thus we are trying to optimize priority subject to TTC and COT constraints. To test this we will use a metric of weighted priority over the output list. This weights the priority of machines closer to the front of the list higher than those at the end.

$$WeightedPriority = \sum_{i=1}^{m} \left(P_i \cdot \lambda^i\right)$$

Where $0 < \lambda < 1$ is the weighting parameter.

It is important to ensure that our makespan and idle times are not severely harmed in the process. The makespan is the total time for all the machines to run and be serviced. The idle time is the time a completed machine must wait before being serviced. Our algorithm is tested against a first-come-first-serve baseline where machines are sorted by TTC.

To achieve this we will define the following constraints.

- *The schedule can be broken into smaller lists of ascending priority. Eg: (in Figure 2), M_2, M_5, M_1 form one of these lists.*
- *For each machine M_i and another machine M_k in each smaller list, if $k < i$ then $COT_k + TTC_k < TTC_i$.*
- *For each machine M_i in each smaller list the sum of $COT + TTC$ of all machines scheduled before M_i must be less than TTC_i. In other words:*

$$\sum_{j=1}^{i-1} (COT_j + TTC_j) < TTC_i$$

These constraints will let us define our knapsack capacity and weights. This will be done dynamically by taking a specific machine's TTC as the capacity and the other machines $COT + TTC$ as the weights. We introduce a notation $x.y$ such that *y* is the attribute of object x *(e.g., $M.TTC$ implies the TTC of machine M).*

Algorithm

The algorithm works as follows:

1. Sort the list of machines by highest *P*, then by lowest TTC then by lowest COT.
2. Select pivot machine u with the highest P and the lowest TTC. Set C to $u.TTC$. Initialize L as an empty list.
3. For each machine v with lower priority than u, in the same order as the input list:
   I. if $v.TTC + v.COT < C$ then append v to L, and subtract $v.TTC + v.COT$ from C.
4. Repeat from 2 on L and also the elements not in L separately until all machines have been scheduled, placing the elements in the first list before u and the elements in the other list after u.

The idea behind this algorithm is that by first selecting machines with the highest priority as our pivot machine, we can then place machines before this pivot so long as they will all complete before the pivot needs to be serviced. If they will not complete in time, they will be placed afterwards. This will ensure that by the time the high-priority pivot machine needs to be serviced, there will be an operator available. Figure 2 gives a case example showing the pivot machines and where the other machines in its list fall.

Algorithm 1.2 GreedyMachineList(m,M) – A greedy approach to selecting which elements should come before machine m

Require: Currently selected machine m, List M of machines to choose from w/ priority less than m's

```
if len(M) == 0 then
  return [m]
else if len(M) == 1 then
  /* If M has exactly one element compare to m and return appropriately */
  if (M0.COT + M0.TTC) ≤ m.TTC then
    M.append(m)
    return M
  else
    return [m]
L = [ ]
capacity = m.TTC
/* Fit machines into capacity */
for x in M do
  if x.TTC + x.COT ≤ capacity then
    L.append(x)
    capacity− = x.TTC + x.COT
OutList = [ ]
if L is not Empty then
  nextPivot = machine with largest priority and smallest TTC in L
  remove nextPivot from L
  OutList = GreedyMachineList(nextPivot,SubList(nextPivot, L))
  for l in OutList do
    if l is in L then
      L.remove(l)
for l in L do
  append l to OutList
append m to OutList
return OutList
```

Figure 1: Partial psuedocode for main portion of the greedy-approach knapsack solution. An additional loop is required with this function.

	M_1	M_2	M_3	M_4	M_5	M_6	M_7
COT	3	1	2	3	1	4	2
TTC	8	2	3	6	3	12	5
Priority	5	1	3	2	4	5	3

Sorted list		Scheduled before	Pivot	Scheduled After	Schedule
M_1	1:	$[M_5,M_2]$	M_1	$[M_6,M_3,M_7,M_4]$	M_1
M_6	2:	$[M_2]$	M_5	[]	M_5,M_1
M_5	3:	[]	M_2	[]	M_2,M_5,M_1
M_3	4:	$[M_3,M_7]$	M_6	$[M_4]$	M_2,M_5,M_1,M_6
M_7	5:	[]	M_3	$[M_7]$	M_2,M_5,M_1,M_3,M_6
M_4	6:	[]	M_7	[]	M_2,M_5,M_1,M_3,M_7,M_6
M_2	7:	[]	M_4	[]	$M_2,M_5,M_1,M_3,M_7,M_6,M_4$

Figure 2: Example set of machines and the algorithm in process.

Results

This algorithm was tested in a Python3 Jupyter-Notebook environment on randomly generated sets of 40 machines. Machines were generated as 3-tuples – (COT, TTC, P) using random values.

From the randomly generated list of machines we constructed a baseline schedule that sorted all machines by TTC. This gives us a good way to compare our algorithm to the first-come-first serve situation which is the basis for the problem. We ran 500 tests per metric and used a λ = 0.9 for weighted priority.

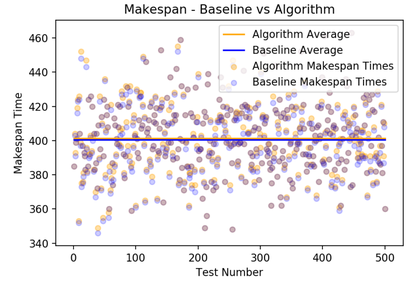

Figure 3: Baseline comparisons on 3 metrics.

From these tests we can see that there is minimal loss to the makespan. This loss occurs as there may be some machines with lower P and TTC that get scheduled after another machine. From Figure 2, we can see this with how M_1 has a $TTC + COT$ greater than the TTC for M_3. We do see a significant improvement on the idle time, which in our circumstance is more important than makespan. Lastly we also see a significant improvement in the weighted priority score, which is what we were looking to achieve.

Prior to any optimization, the time complexity of this algorithm runs at a worst case of $O(m^2)$. Since the machine list will regularly need to be updated and requires sorting, our sorting becomes a lower bound of $O(m \log m)$ with the quicksort algorithm.

Conclusions

While there is an increase to makespan time, the large reduction in idle time and increase in weighted priority indicate a strong argument for the usage of this algorithm. One of the major draw backs of this approach is that machines with low priority and high change-over time may never be scheduled. For example if all machines have a max TTC of 10 (the full cycle time of machines is no more than 10) and machine m has a COT of 11 machine m will always be scheduled last. However, we have already devised ways to handle these situations by allowing TTC become negative if the machine sits idle, and using activation functions such as Rectified Linear Units to weight this negative descent.

The worst-case time complexity for the algorithm of $O(m^2)$ could cause problems. On one hand, in most circumstances, we will have relatively few machines. However, many of the applications of this algorithm are real-time and thus efficiency is important. Further analysis may allow us to bring this down by a constant factor, or possibly even to $O(m \log m)$ if possible.

This algorithm will have applications in any situation where trying to optimize some parameters subject to time constraints with time delay. For example in a computer system where many programs a vying for a resource, like a finite number of data lines, and holding that resource for some amount of time, before processing the data independently. We could then use this algorithm to optimize the priority assigned to each program.

Future Research

Our goal will be to explore other optimization approaches such as Lagrange Multipliers, and the aforementioned Leader Election problem approach. Since the algorithm mentioned here is designed with only one operator and assumes that this will extend well to many operators, we may look at ways of improving this algorithm further if the number of operators is known. We will also look into ways of improving the time-complexity of this algorithm, and applying other knapsack problem solutions.

Acknowledgements

Thank you to Dr. Adesina for the opportunity to work on this project, and the amazing advice and support he has given throughout. Thank you also to **Harmonic Machine Inc.** for giving us the opportunity to work on this problem and present results of this research.

UNIVERSITY OF THE FRASER VALLEY