Scheduling Programming Activities and Johnson's Algorithm

Allan Glaser and Meenal Sinha
Octagon Research Solutions, Inc.

Abstract

Scheduling is important. Much of our daily work requires us to juggle multiple activities in order to maximize productivity. In general, we strive to minimize the overall time required to complete a series of related tasks. Although this is a very complex topic, there are certain situations that lend themselves to closed form solutions. Johnson's algorithm is one approach that can often be applied. The algorithm is easily stated and understood, and can be readily implemented in SAS®. In addition to directly helping with scheduling, the programming illustrates some interesting SAS capabilities, and the final code is very simple and straightforward.

Background and Introduction

The problem will be introduced with a small example. Two members of a programming department have the assignment of producing a set of six programs. One person will do the initial programming, and the other person will validate the work. The programming and validation activities are independent, but obviously the validation must follow the programming. The complete set of 6 programs will be delivered as a unit, but the individual programs can be completed in any order. Here are the programs with the expected programming and validation times (in consistent but arbitrary units):

program	programming time	validation time
A	15	4
B	22	10
C	3	6
D	12	8
E	9	6
F	12	9

In what order should the work be done to minimize the overall elapsed time?

Estimating Programming Effort

Effort estimates for programming can vary due to inherent assumptions being made about (a) the scope and/or definition of work, (b) programmers' skills and experience, and (c) availability of tools, test data and computing environments. While any set of estimates can be used for planning, the degree of confidence in the plan can be substantially increased by assimilating the expected time concept into the estimation process.

Based on PERT/CPM project planning techniques, the expected time blends a variety of estimates into one robust estimate that is more reliable. Expected time (E) is based on optimistic time (O), most likely time (M), and pessimistic time (P) as follows:

$$E = (O + 4M + P) / 6$$

This expected time (E) for each activity should be used as the best estimate to be plugged into the scheduling algorithm described here.

Problem Details

The overall elapsed time between the start of the first programming activity and the completion of the last validation activity is called makespan. The objective of scheduling is to find an optimal order for the programming and validation, and thus minimize the makespan. Determining an optimal schedule requires some reasonable assumptions:

- The programming can only be done on one program at a time.
- The programming must be completed before validation can begin.
- Validation can only be done on one program at a time.
- The order of completing these programs is not important.
- After the initial programming, validation may be deferred.
- The validation order is the same as the programming order.
- There are no priorities or dependencies among these programs.

Johnson's algorithm gives us a straightforward approach to finding an optimal order. There may be multiple orderings that yield the same minimal makespan; Johnson's algorithm will identify just one solution.

It is important to understand that although this simple scenario may be valid in many real-world situations, there will be many problems that cannot be sufficiently simplified to fit Johnson's approach.

Even with this small example, the optimal order is not obvious. Some people may intuitively schedule the most time-consuming activity first; the B-A-F-D-E-C order will yield a makespan of 82 units of time. Others may prefer the reverse order, i.e. C-E-D-F-A-B, thinking it wise to actually finish some work as early as possible. That will result in a makespan of 83. In this small example, however, the optimal makespan is actually 77, which can be achieved by scheduling in C-B-F-D-E-A order.

At first glance, the difference between 82 or 83 and the optimal 77 may seem small and unimportant. But assuming that there are no significant short-term activities that can be used to fill any gaps in the schedule, it reflects a change in overall throughput and productivity of roughly 7%. In most programming shops, that is a very substantial improvement.

Johnson's Algorithm – A Closer Look

In generic terms, our sample problem consists of a set of independent tasks, namely T_1 (program A), T_2 (program B), etc., and two independent processes, P_1 (programming) and P_2 (validation). The process order is fixed – P_1 must be completed before P_2 can start. Johnson's algorithm is simple:

1. For each task T_1, T_2, ..., T_n, determine the P_1 and P_2 times.
2. Establish two queues, Q_1 at the beginning of the schedule and Q_2 at the end of the schedule.
3. Examine all the P_1 and P_2 times, and determine the smallest.
4. If the smallest time is P_1, then insert the corresponding task at the end of Q_1. Otherwise, the smallest time is P_2, and the corresponding task is inserted at the beginning of Q_2. In case of a tie between a P_1 and P_2 time, use the P_1 time.
5. Delete that task from further consideration.
6. Repeat steps 3 – 5 until all tasks have been assigned.

What happens if there is a tie between multiple P_1 or multiple P_2 times? There is no reason to prefer one task to another in this situation, and simply selecting the first task in the list is sufficient.

This algorithm is easy to use manually, but there are advantages to implementing it in a SAS program. As tasks are completed, and as process times are revised, a program allows a new schedule to be created with minimal effort. A program also allows experimenting and looking at the results when a critical task is manipulated. And, having a systematic approach with scheduling encourages its use.

Programming the Algorithm

Multiple programming approaches were initially considered. In all of these, the first step began with getting the raw data into a SAS dataset:

TASK	PROCESS1	PROCESS2
A	15	4
B	22	10
C	3	6
D	12	8
E	9	6
F	12	9

The next step was to determine the smaller process time for each task, and also note if that time came from the first or second process. Tasks with P_1 time less than P_2 time would be assigned to the head queue Q_1 ; other tasks would be assigned to the tail queue Q_2. This dataset results:

TASK	PROCESS1	PROCESS2	MINTIME	PLACE
A	15	4	4	tail
B	22	10	10	tail
C	3	6	3	head
D	12	8	8	tail
E	9	6	6	tail
F	12	9	9	tail

Lastly, the dataset was sorted by MINTIME, yielding:

TASK	PROCESS1	PROCESS2	MINTIME	PLACE
C	3	6	3	head
A	15	4	4	tail
E	9	6	6	tail
D	12	8	8	tail
F	12	9	9	tail
B	22	10	10	tail

The first programming approach created two stacks. These were implemented as arrays and correspond to Q_1 and Q_2. As each observation was read, the project was pushed to the appropriate stack. After the last project was assigned the stacks were combined – Q_2 was inverted and appended to Q_1. Although this worked, it required a significant amount of code.

The second programming approach used a linked list to represent the schedule. This was also implemented with an array, and included pointers to preceeding and subsequent list entries. This worked, but like the stack approach, required a significant amount of code.

Occam's razor reminds us that the simplest approach may be the best, and that was true here. The final approach was quite simple. Two character variables are defined, corresponding to Q_1 and Q_2. As each observation is read from the sorted dataset, if PLACE has the value "head", then the task is concatenated to the end of Q_1. Conversely, if PLACE has the value "tail", then the task is concatenated to the front of Q_2. After the last task has been assigned, HEAD and TAIL are concatenated, thus creating the final complete list in variable QUEUE. In this example, the value of QUEUE is "C B F D E A". Some code changes are needed, of course, to handle longer task names. Replicating this list in an array or in separate observations in another dataset is trivial. Here is the core code:

```
data queue ;
   attrib queue head tail length=$200 ;
   retain head tail  '' ;
   set raw end=lastrec ;
   if place = 'head' then head = trim(head) || ' ' || task ;
      else                tail = trim(task) || ' ' || tail ;
   if lastrec then do ;
      queue = trim(head) || ' ' || tail ;
      put queue= ;
      end ;
run ;
```

Efficiency

Program efficiency is always a consideration. Scheduling programming shop activities will normally involve a small amount of data, and applying Johnson's algorithm should require minimal computer resources. Nonetheless, it may be instructive to look at some details.

In each of the three above approaches, the data are stepped through one observation at a time and the data must be sorted. The time to do the former depends only upon the volume of data, and thus can be done in O(n) time. The sorting, however, requires O(n*log(n)) time, and that term dominates. Thus, actually using Johnson's algorithm and determining an optimal schedule is a function of n*log(n).

Calculation of Makespan

The calculation of makespan is interesting. At first glance, it is simply the sum of the times for the first process, plus any second process times that extend beyond the end of the first process for the last task. Looking at the optimal solution for our sample problem, we begin by noting that the first process can continue without interruption, yielding:

TASK	PROCESS1	PROCESS2	PROCESS 1 START TIME	PROCESS 1 STOP TIME
C	3	6	0	3
B	22	10	3	25
F	12	9	25	37
D	12	8	37	49
E	9	6	49	58
A	15	4	58	73

Since the first process is in continuous use for 73 units of time, and the second process for the last task requires 4 units of time, the makespan must be at least 73 + 4 = 77. In this example, the second process must frequently pause while it waits for a task to be completed by the first process. If the converse were true, then the makespan would be minimally the sum of the second process times, plus the first task's time for the first process.

The calculation of makespan in realistic situations is beyond the scope of this paper, but it frequently helps to display everything in a graphical or Gantt chart form. PROC TIMEPLOT is a very simple way to do this:

```
 task   begin   stop_1   finish     min                                      max
                                     0                                        84
                                     *-----------------------------------------*
  C        0        3        9      |BX--F                                    |
  B        3       25       35      | B----------X----F                       |
  F       25       37       46      |            B-----X----F                 |
  D       37       49       57      |                  B-----X---F            |
  E       49       58       64      |                        B----X--F        |
  A       58       73       77      |                             B------X-F  |
                                     *-----------------------------------------*
```

Discussion

What happens if there are three or more processes? In a SAS programming shop, these processes may be writing specifications, programming, and validation. This situation does not have a straightforward solution. In practice, solutions take the same general approach as the two-process situation, i.e. as tasks move through multiple processes, priority is given to tasks with the greatest tendency to move from short times to long times. There are numerous algorithms and approaches for doing this, and there is evidence that the solutions are near-optimal, but it is not clear how to quantify any differences. Furthermore, these algorithms may allow the tasks to progress through the different processes in different orders.

Another potential complication is interference. Interference is defined as having conflicting demands on a single process, e.g. a single programmer being instructed to simultaneously work on multiple high-priority tasks, but able to work on only a single task at a time.

Yet another complication is having multiple resources for each process. In these situations, which may include contemporary programming techniques such as pair programming, it is frequently helpful to simply consider the elapsed time that multiple resources will each concurrently spend on a process. In other words, if multiple programmers are working on the same task at the same time, simply consider the elapsed time, and not the total person-time.

Note on SAS/OR® for Operations Research

The SAS/OR package does provide sophisticated scheduling capabilities for very large manufacturing, construction and logistics enterprises.

SAS/OR software's project management capabilities gives the flexibility to plan, manage and track project and resource schedules through a single integrated system. The software is adept at handling complicated situations involving multiple project record keeping, resource priorities, complex project and resource calendars, substitutable resources with skill pools, multiple and nonstandard precedence relationships, and activity deadlines.

Implementing SAS/OR for Clinical Programming departments will be overkill. The simpler scheduling algorithm and program suggested in this paper are easier to implement and use in most cases.

Conclusion

Our typical day-to-day work does not completely conform to Johnson's assumptions. We generally must deal with assigned priorities, dependence among tasks, and processes which can concurrently handle multiple tasks. Nevertheless, Johnson's algorithm can be helpful and can frequently add insight to help planning, and it is essential that we understand the importance of minimizing makespan.

Implementing Johnson's algorithm in SAS is deceptively simple. Initial approaches that were complex were abandoned in favor of a robust and easy technique. Many real world programming scheduling scenarios can be simplified to use the resulting program, and that can yield valuable insight to help programmers and managers schedule work to help optimize productivity.

Acknowledgments

The authors thank coworkers who share an interest in scheduling. Their constructive comments and enlightened discussion are appreciated.

References

Garg N, Jain S, & Swamy C. A Randomized Algorithm for Flow Shop Scheduling. *Proceedings of FSTTCS*, 1999, 213-218. This gives valuable insight for complex scheduling situations, but does not offer complete practical solutions.

Hong TP, Chuang TN. Fuzzy Palmer Scheduling for Flow Shops with More Than Two Machines. *Journal of Information Science and Engineering*, 1999, 397-406. This addresses Palmer's work more rigorously.

Johnson SM. Optimal Two- and Three-Stage Production Schedules with Setup Times Included. *Naval Research Logistics Quarterly*, 1954, March: 61-68. This is the first paper that solved the 2-process problem. It also addresses some issues with the 3-process problem.

Palmer DS. Sequencing Jobs through a Multi-Stage Process in the Minimum Total Time – A Quick Method of Obtaining a Near Optimum. *Operational Research Quarterly*, 1965, March: 101-107. This paper establishes the rule for prioritizing tasks that have increasing times for later processes.

Sasieni, Yaspan, & Friedman. *Operations Research*, 250-269. Wiley; 1959. This is a detailed exploration of Johnson's original paper.

SAS/OR. http://www.sas.com/technologies/analytics/optimization/or/#section=1. This gives an overview of the SAS/OR product and capabilities.

Allan Glaser is a manager and Meenal Sinha is a Senior Associate in the Clinical Programming department at Octagon Research Solutions, Inc., and can be contacted at aglaser and msinha, both at @octagonresearch.com. Comments regarding this work would be appreciated.