Research Journal of Mathematics and Statistic 1(1): 4-8, 2009
ISSN: 2040-7505

Submitted Date: June 02, 2009 Accepted Date: July 16, 2009 Published Date: September 23, 2009

Comparison of the Running Time of Some Flow Shop Scheduling Algorithms using the Big O Notation Method

[1]E.O. Oyetunji and [2]A.E. Oluleye
[1]Department of Applied Mathematics and Computer Science
University for Development Studies, Ghana
[2]Department of Industrial and Production Engineering University of Ibadan, Nigeria

Abstract: The quality of solution obtained by scheduling algorithms can be evaluated in terms of both effectiveness (a measure of the closeness of the value of the objective function to the optimal) and efficiency (a measure of the execution speed or running time). This paper focused on the later. We showed how the running time of an algorithm can be calculated using the Big O notation method. Since sorting techniques form the building blocks of many algorithms, the running time of some of the commonest ones were also calculated. The running time of some flow shop heuristics proposed long ago were estimated and used as a basis of comparing their efficiencies.

Keywords: Efficiency, Big O notation, algorithms, scheduling, flow shop

INTRODUCTION

Scheduling is concerned with the problem of allocating resources (machines) over time in order to perform a number of tasks (jobs) with the aim of minimizing cost or maximizing profit (Oyetunji and Oluleye, 2007). The problem is essentially a decision-making problem. The order in which tasks (jobs) are performed on the machine(s) is of interest to the decision maker (Oyetunji, 2006). Usually, the order adopted affects the value of an objective (performance) measure. The problem is to determine the optimum ordering (sequence) which best meet the set objectives.

There are basically two methods of solving scheduling problems. These are: exact and approximation methods (Uthaisombut, 2000, Oyetunji, 2006). Exact method yield optimal solutions (e.g. total enumeration method, Hungarian method, Johnson's method for 2-machine sequencing, implicit enumeration method such as branch and bound or dynamic programming methods).

The approximation method, on the other hand, involves the use of heuristic algorithms. Heuristic methods usually involve the use of intuitive approach or rule of thumb. The heuristic methods are techniques for obtaining acceptable solutions to scheduling problems at a reasonable computational cost (Oyetunji and Oluleye, 2007). While they do not always guarantee optimal results, the techniques are relatively economical in terms of computational resources utilized. As a result of the difficulties and computational complexities involved in obtaining an optimal solution to scheduling problems (even the simplest form of the single machine case), research efforts have been directed towards the development of constant-factor approximation algorithms. A constant-factor approximation algorithm is one that produces a solution whose value is always within a certain constant factor of the optimal value.

At present, many approximation algorithms have been proposed for different classes of scheduling problems (Oyetunji and Oluleye, 2008). The performances of some of the approximation algorithms have been evaluated based on their effectiveness (a measure of the closeness of the value of the objective function to the optimal) without any mention of their efficiency (a measure of the execution speed or running time). Few researchers have mentioned the running time of some algorithms in passing without actually showing how to calculate the running time of such algorithms.

In this work, this equally vital aspect of performance evaluation of scheduling algorithms is addressed. We showed how the running time of an algorithm can be calculated using the *Big O notation* method. Since sorting techniques or methods form the building block of many algorithms, the running time of some of the commonest sorting methods were first calculated. The paper went further to estimate the running time of some flow shop heuristics proposed long ago, thereby providing opportunity to compare their performances in terms of their efficiencies.

MATERIALS AND METHODS

Basis of evaluating efficiency of algorithms: It is often desired to choose which algorithm is more efficient than other. To be able to do this, there is need to have some common basis for evaluating the efficiency of the algorithms. The following are some of the common basis that can be used:

Corresponding Author: Dr. E.O. Oyetunji, Department of Applied Mathematics and Computer Science University for Development Studies, Ghana

(a)The same class of problem: There are many classes of scheduling problems. For example, we have flow shops problems (two machine flow shops, three machines flow shops etc.), job shop problems, and open shops problems. Problems can also be classified in terms of the objective function: single criterion problems, bicriteria problems, multi criteria problems etc. Problem classification can also be in terms of the job characteristics such as problems that allows pre-emption, release dates constraints, precedence constraints etc. The issue here is that, only algorithms that attempt to solve the same class of scheduling problems (does the same thing) may be compared together.

(b)The same instance of problem: By this we mean a particular occurrence of a scheduling problem with particular problem data. There could be real life or randomly generated problems. Whether it is real life or random problems, only algorithms that are applied to a particular problem data (instance) may be compared together. This is to ensure that the peculiarity of scheduling problem data does not affect the running time of the algorithms being evaluated.

(c)The same machine or computer: Because of the differences in the speed of operation of the different computer processors, it is advisable to test the algorithms on the same type of computer system and same processor type. This third point is a little bit weak because the Big O notation which we shall soon discuss has taken care of the differences in the computer speed.

Calculating the running time of an algorithm: The running time of algorithm can be defined as the time that the algorithm requires to solve a given problem (Dale *et al*., 2002). It can also be defined as the number of steps an algorithm requires to solve a specific problem. It is, usually, expressed in relation to input size (n). The running time of an algorithm will typically change depending on the size of input. Algorithms have a best case, average case, and worse case running time. The worst case running time is the longest possible time it will take the algorithm to solve the given problem. The best case is the shortest possible time it will take the algorithm to solve the given problem, while the average case attempts to estimate the average time that the algorithm will take given a certain size of input. The actual running time of an algorithm can be measured by taking time measurements at the beginning and end of the algorithm execution. This is dependent on the type of hardware used. However, a reasonable measurement of running time should be machine independent. The time required by an algorithm is proportional to the number of basic operations that it performs (Aho *et al*., 2003).

In calculating the running time of an algorithm, we do not really care about the exact number of operations that are performed; instead, we care about how the number of operations relates to the problem (input) size. For example, if the problem size doubles, does the number of operations stay the same? double? or increase in some other way? For constant-time algorithms, doubling the problem size does not affect the number of operations (which stays the same). One common and acceptable method of estimating the running time of an algorithm is the Big- O notation (also called order of magnitude) method.

The Big O notation: The Big O notation is a theoretical measure of the execution time of an algorithm (Dale *et al*., 2002). It is an approximate way to express the complexity of an algorithm, where only the most rapidly increasing factor is shown. The Big O notation is often used to describe how the size of the input data affects an algorithm's usage of computational resources (usually the running time). It is also called Big Oh notation, Landau notation, and *asymptotic notation*. Big O notation is used in many scientific and mathematical fields to provide similar estimations.

Big O notation focuses on the growth rate of running time of an algorithm as the input size (n) approaches infinity. It gives us a formal way of expressing asymptotic upper bounds, a way of bounding the growth of a function or algorithm. Knowing where an algorithm falls within the big-Oh hierarchy allows us to compare it quickly with other algorithms, which gives us an idea of which algorithm has the best time performance (Parrilo and Lall 2003). The Big O notation method is an acceptable way of estimating/calculating the order of magnitude or the running time of an algorithm (Haipeng and William 2004).

Some rules in estimating the Big O notation: To be able to correctly estimate the order of magnitude of an algorithm, certain rules must be followed (Dale *et al*., 2002). These rules are discussed below.

(a) Sum rule: The sum rule states that the running time of an algorithm is the sum of the running time of the various segments (to be executed sequentially) that made up the algorithm. For example, suppose that an algorithm has two segments or fragments, labeled A_1 and A_2. If the running time of A_1 and A_2 are $T_1(n)$ and $T_2(n)$ respectively, then the running time of algorithm A (say $T(n)$) is $T_1(n) + T_2(n)$. Therefore, the order of magnitude of A is $O(A(n)) = O(\max(T_1(n) , T_2(n)))$.

(b) Product rule: The product rule states that the running time of an algorithm is the product of the running time of the various segments (to be executed within each other) that made up the algorithm. For example, if segment A_2 above is an inner loop to be executed inside an outer loop A_1, then the running time of algorithm A ($T(n)$) is $T_1(n)T_2(n)$. Therefore, the order of magnitude of A is $O(A(n)) = O(T_1(n)T_2(n))$.

Generally, the keys to successful estimation of the running time of an algorithm are as follows:

- The running time of any basic computer operation such as an assignment is usually taken to be of order O(1).
- Use the sum rule to determine the running time of a sequence of operations (operations that are performed one after the other).
- Use the product rule to determine the running time of nested loops (operations).

RESULTS AND DISCUSSION

Order of magnitude of some sorting techniques: Since many algorithms usually employ one sorting technique or the other, we shall estimate the running time of some of the commonest sorting techniques.

Suppose we are given a list of n data items (no of jobs) that are to be sorted in a particular order (ascending or descending order).

(a) Straight selection sort: Selection sort is implemented with one loop nested inside another. The first iteration through the data requires *n* - 1 comparisons to find the minimum value to swap into the first position. Because the first position can then be ignored when finding the next smallest value, the second iteration requires n-2 comparisons and third requires n-3. The last requires 1 comparison. The worse case is that the first iteration which requires n-1 comparisons will be executed n times (Aho *et al*., 2003). Therefore, using the product rule (since this is a case of a nested loop; one loop inside another), the total number of comparisons is given as: n(n-1). Therefore, in terms of the Big-O or order of magnitude notation (also called the time complexity), the straight selection sort technique is an $O(n^2)$ algorithm since n has little effect on n^2 when n is very large (as n approaches infinity). For example, when n = 2, $n^2 = 4$, the difference between n^2 and n is not much. However, if n = 500, then $n^2 = 250{,}000$, in this case the difference is much. So as the value of n increases, the n^2 component dominates the n component, hence the straight selection sort technique is $O(n^2)$.

(b) Bubble Sort: The bubble sort technique requires a pair of nested loops. The outer loop iterates once for each element in the data list while the inner loop iterates n times the first time it is entered, n-1 times the second, n-2 times the third, and so on. Bubble sort is structured so that on each pass through the list the next largest element of the data list is moved to its proper place. Therefore, for us to get all the n elements in their correct places, the outer loop must be executed n times (Aho *et al*., 2003).

The inner loop is executed on each iteration of the outer loop. Its purpose is to put the next largest element into place. The inner loop therefore does the comparing and swapping of adjacent elements. To determine the complexity of this loop, we calculate the number of comparisons that have to be made. On the first iteration of the outer loop, while trying to place the largest element, there have to be n - 1 comparisons: the first comparison is made between the first and second elements, the second is made between the second and third elements, and so on until the n-1th comparison is made between the n-1th and the nth element. On the second iteration of the outer loop, there is no need to do comparison against the last element of the list, because it was put in the correct place on the previous pass. Therefore, the second iteration requires only n-2 comparisons. This pattern continues until the second-to-last iteration of the outer loop when only the first two elements of the list are unsorted; clearly in this case, only one comparison is necessary. Again, the worse case is that the inner loop iterates n times while the outer loop also iterates n times. The total number of comparisons (using the product rule), is given as: n(n). Therefore, the bubble sort technique is $O(n^2)$.

(c) Merge sort: To understand the order of magnitude (time complexity) of the merge sort technique it is useful to separate the merging from the sorting. The sorting takes place indirectly, by repeatedly splitting the data in half until sorted singleton sets are created (Aho *et al*., 2003). The merging then rebuilds the complete, original data set by splicing together the sorted mini-lists. To determine the time complexity of the sorting (breaking down), consider how many times the data has to be split. A data set of size 4 has to be split twice, once into two sets of two and then again into four sets of one. A data set of size 8 has to be split 3 times, 16 pieces of data have to be split 4 times, 32 needs 5 splits, and so on. This sort of behavior is reflected by the logarithm:

- log2(4) = 2
- log2(8) = 3
- log2(16) = 4
- log2(32) = 5.

The breaking down of the data, then, occurs with time complexity O(log n). The merging process is linear each time two lists have to be merged, because it is simply done by doing one comparison for each pair of elements at the top of each sublist. Therefore, the merge process occurs n times (with time complexity of O(n)). Combining the time complexity for the two processes (using the product rule) gives the time complexity of the merge sort technique which is O(nlog(n)).

(d) Quick sort: The quick sort is a divide-and-conquer algorithm which is inherently recursive (Aho *et al*., 2003, Xiaoming *et al*., 2004). The quick sort technique, usually pick a splitting value and then divide the list being sorted into two piles (left pile and right pile). The items in the left pile are less than the splitting value while the items in the right pile are greater than the splitting value. Each of the two piles is equally (in the same manner) subdivided into two sub piles each until the number of items in each sub piles is two. At the bottom, this resulted in a number of sorted sub piles. Eventually, all the sorted sub piles are collected one on top of the other to produce a sorted list. During the recursive process, every element in the list is compared with the dividing value (splitting value) thereby requiring n comparisons (O(n)). As is common with

divide-and-conquer sorts, the dividing algorithm has a running time of log(*n*) (like the merge sort). Thus, the overall quick sort technique (using the product rule) has time complexity O(nlog(n)).

(e) Heap sort: A heap is a data structure with special feature that allows us to always know where to find its greatest element (Aho *et al.*, 2003). The greatest element of a heap is in its root node. The general approach of the heap sort technique is as follows:

(a) Take the root (maximum) element off the heap and put it in its place
(b) Reheap the remaining elements so as to put the next maximum element into the root position.
(c) Repeat until there are no more elements.

The central task in the heap sort algorithm is restoring the heap after each removal of the root element (step b). Step a above takes constant time (O(1)) irrespective of the value of n. Step b takes log(n) time since there are log(n) levels in the tree that the value may have to move through. Step c will be repeated at most (n-1) times which gives a (n-1)*log(n) time (using the product rule. Using the sum rule, the total time for the heap sort is O((1) + log(n) + (n-1)*log(n)) which give the running time of the heap sort to be O(nlog(n) since nlog(n) dominates other components as n increases.

Common orders of magnitudes: Here is a list of classes of order of magnitudes that are commonly encountered when analyzing algorithms (Table 1). The slower-growing functions are listed first. Note that c is an arbitrary constant.

Table 1: Some Common order of magnitudes

Notation	Name
O (1)	Contant
O (log* n)	Iterated logarithmic
O (log n)	Logarithmic
O ((log n) ^{c})	Poly logarithmic
O (n ^{2}), 0 < c < 1	Fractional power
O (n)	Linear
O (nlogn)	Linearithmic, loglinear, or quasilinear
O (n^{2})	Quadratic
O (n ^{c}) , c > 1	Polynomial, sometimes called algebraic
O (c ^{n})	Exponential, sometimes called geometric
O (n !)	Factorial, sometimes called combinatorial
$O\left(c_1^{c_2^n}\right)$	Factorial, sometimes called combinatorial

Some selected scheduling algorithms:
One class of scheduling problems that have been studied by many researchers is the problem of minimizing makespan in an m machine flow shop situation (French, 1982). A number of heuristics have been proposed for solving this class of scheduling problem and performance evaluation, which focused only on the effectiveness of the heuristics, has been carried out. Four of such heuristics were selected and their running time (order of magnitude) estimated thereby enabling comparisons based on their efficiencies.

(a) Palmer (PAL) Heuristic: Palmer (1965) proposed a heuristic for the general n x m (n = number of jobs, m = number of machines) flow shop problem. His heuristic uses a slope order rule. A numerical slope index was calculated for each job, the sequence is obtained by arranging the jobs in the order of decreasing slope indices. The job with the highest index is scheduled first while the job with the lowest index is scheduled last. The steps for PAL heuristic are as follows:

Step 1: Calculate the slope index for each job.
Step 2: Schedule jobs according to the descending order of the slope index.

Since there are n jobs, step 1 will take O(n*m). Step 2 will involve sorting which can be done in at most O(n^{2}) time. Therefore, total running time of PAL heuristic is O(n*m + n^{2}). If m <= n; then PAL runs in O(n^{2}) time otherwise it runs in O(n*m) time.

(b) Campbel, Dudek and Smith (CDS) Heuristic: Campbell, Dudek and Smith (1970) also considered the general n x m flow shop scheduling problem. They developed a procedure called the CDS heuristic. The algorithm generates (m-1), 2-machine sub-problems for the general n x m flow shop problem. Johnson's 2-machine algorithm is then applied to solve each of the sub-problems. The schedule that gives the best solution is chosen as the solution to the original problem. The CDS steps are as follows:

Step 1: Form m-1 artificial sub problems from the original m- machines problems
Step 2: Apply Johnson's algorithm for 2-machines problem: Johnson's algorithm for 2-machines problem states that if the least processing time is found on the first artificial machine, then schedule the job with the least processing time to early position otherwise if the least processing time is found on the second artificial machine, then schedule the job to later position. This is done iteratively until all the jobs must have been scheduled.
Step 3: Repeat step 2 for all the m-1 artificial sub problems.
Step 4: Choose the best solution out of all the m-1 solutions.

Since there are n jobs and m machines, step 1 will be executed (using the product rule) in O(n(m-1)) time. Step 2 will require two loop to schedule all the jobs, therefore, it will take O(n^{2}). Step 3 will be repeated m-1 times (that is once for each of the sub problems), it will take (using the product rule) O(n^{2}(m-1)). Therefore, using the sum rule, the running time of the CDS heuristic is given as O(n(m-1) + n^{2} + n^{2}(m-1)). It is obvious the term n^{2}(m-1) has a higher growth rate in this function, therefore the CDS heuristic will run in O(n^{2}(m-1)) time. For a special case where m=n, CDS will run in O(n^{3}) time.

(c) Dannenbring (DAN) Heuristic: Dannenbring (1977) proposed a flow shop sequencing heuristic that uses a weighting scheme similar to that of Palmer and Campbell, Dudek and Smith to form a single 2-machine artificial sub problem. The resulting 2-machine sub problem is then solved by applying Johnson's 2-machine algorithm. The DAN steps are as follows:

Step 1: Form a single artificial 2-machine sub problem from the original m- machines problems
Step 2: Apply Johnson's algorithm for 2-machines problem (as stated above)
Step 3: Stop.

Step 1 will be executed (using the product rule) in O(n*m) time. Step 2 (as in the above) will require two loop to schedule all the jobs, therefore, it will take $O(n^2)$. Therefore, using the sum rule, the running time of the DAN heuristic is given as $O(n*m + n^2)$. When m <= n, the DAN heuristic will run in $O(n^2)$ time while it runs in O(n*m) time when m > n.

(d) A1 Heuristic: Oluleye and Oyetunji (1999) studied the n x m general flow shop problem. They proposed a heuristics called A1 which also forms (m-1), 2-machine artificial sub problems. Johnson's 2-machine algorithm is then applied to solve each of the m-1 artificial sub problems. The schedule that gives the best solution is chosen as the solution to the original problem. The difference between A1 and CDS is in the way the artificial sub problems are formed. Therefore, the A1 heuristic also has $O(n^2(m-1))$ time complexity. And for a special case where m = n, A1 will run in $O(n^3)$ time.

Comparison of efficiency of the selected algorithms: In comparing the efficiency of the selected heuristics, let us assume that m = n. The running time of the four selected heuristics are summarized in Table 2. Based on the *order of magnitude* (time complexity) of the heuristics, it is obvious that both PAL and DAN heuristics are more efficient than the CDS and A1 heuristics.

Table 2: Running time of selected heuristics

Heuristics	Running time (for special case m = n)
PAL	$O(n^2)$
DAN	$O(n^2)$
CDS	$O(n^3)$
A1	$O(n^3)$

CONCLUSION

The Big O notation method is an acceptable way of estimating/calculating the order of magnitude or the running time of an algorithm. It gives an idea of the running time that is completely independent of the hardware. The running time of some sorting techniques estimated is a pointer to researchers to know what techniques to adopt when designing algorithms since many of these sorting techniques form the building blocks for many scheduling algorithms. From the analysis of the running time of the sorting techniques carried out, it is recommended that either of the merge, quick or heap sort techniques may be used since they all have better time complexity (O(nlog(n))). Also, the result of the comparisons carried out on four flow shop heuristics (PAL, DAN, CDS and A1) showed that the PAL and DAN heuristics are more efficient than the CDS and A1 heuristics (Table 2).

REFERENCES

Aho, A.V., J.E. Hopcreft, and J.D. Ullman, 2003. Data Structures and Algorithms, Pearson Education (Singapore) Pte. Ltd., Delhi, India.

Campbell, H.G., R.A. Dudek and M.L. Smith, 1970. A Heuristic Algorithm for the n Job, m machine Sequencing Problem. Management Science 16: B630-B637.

Dale, N., D.T. Joyce and C. Weems, 2002. Object-Oriented Data Structures using Java, Jones and Barlett Publishers, inc.

Dannenbring, D.G., 1977. An evaluation of flowshop sequencing heuristics. Manage. Sci., 23:1174-1182.

French, S., 1982. Sequencing and Scheduling, Ellis Horwood Limited, ISBN 0-85312-364-0.

Haipeng, G. and H.H. William, 2004. A Learning-Based Algorithm Selection Meta- reasoner for the Real-Time MPE Problem. Australian Conference on Artificial Intelligence, pp: 307-318. http://ieeexplore.ieee.org/xpl/ freeabs_all.jsp ?arnumber=1281668

Johnson, S.M., 1954. Optimal two- and three- stage production schedules with setup times included. Nav. Res. Log., 1: 61-68.

Oluleye, A.E. and E.O. Oyetunji, 1999. Performance of some static flowshop scheduling heuristics. Direct. Math., pp: 315-327.

Oyetunji, E.O., 2006. The Development of Scheduling Heuristics for the Bicriteria problems on a Single Machine, PhD thesis, University of Ibadan, Nigeria.

Oyetunji, E.O. and A.E. Oluleye, 2007. Heuristics for minimizing total completion time on single machine with release time, Adv. Mater. Res., Trans Tech Publications Ltd., Switzerland, 18-19: 347-352.

Oyetunji, E.O. and A.E. Oluleye, 2008. Heuristics for minimizing total completion time and number of tardy jobs simultaneously on single machine with release time. Res. J. Appl. Sci., 3(2): 147-152.

Palmer, D.S., 1965. Sequencing jobs through a multi-stage process in the minimum total time-a quick method of obtaining a near optimum. Oper. Res. Quart., 16: 101-107.

Parrilo, P. and S. Lall, 2003. Semidefinite Programming Relaxations and Algebraic Optimization in Control. IEEE Conference on Decision and Control, 8, December.

Uthaisombut, P., 2000. New Directions in Machine Scheduling, PhD thesis, Michigan State University.

Xiaoming,Li, M.J. Garzaran and D.Padua, 2004. A Dynamically Tuned Sorting Library. Proceeding of International Symposium on Code Generation and Optimization (CGO'04), March 20-24:111-122.