Solution algorithms for a deterministic replacement problem

Zakir H. Ahmed zhahmed@gmail.com
Department of Computer Science,
Al-Imam Muhammad Ibn Saud Islamic University,
P.O. Box No. 5701, Riyadh-11432
Kingdom of Saudi Arabia

Abstract

We consider the life of a k-series system with n-k standbys, which are to replace the working items as they fail in a specified order. Also the life of each item is assumed to be known. The problem is to find the first k elements to be used in the system to start with and the sequence in which replacements are to be made such that the system life is maximized. It is pointed out that this problem is essentially a 'bottle-neck' problem in that it is equivalent to partitioning n numbers into k subsets (parts) such that the minimum of the k part-sums is maximized. Effective bounds can be computed for the optimum solution value and this can be used to develop an efficient lexisearch algorithm for optimization. Also the paper develops a genetic algorithm which gives heuristically optimal solution to the problem. The efficiency of the genetic algorithm to the 'bottle-neck' maximin problem as against lexisearch algorithm has been examined for some randomly generated instances of different sizes.

Keywords: Deterministic replacement, bottleneck, lexisearch, genetic algorithms.

1. INTRODUCTION

In reliability engineering, the k-out-of-n series system is an important system as most systems can be modeled as series system. It is defined to a complex coherent system with n independent components such that the system operates if and only if at least k of these components function successfully. For a complex and expensive system, it may not be advisable to replace the system just because of the failure of one component. In fact, the system re-operates on repair or replacement of the failed component by a new one. Such replacement does not renew the system, but enable the system to continue to operate. So, the system can re-operate as long as number of failed components does not exceed n-k. However, once the number of failed components surpasses n-k, the system does not re-operate [16]. This system is also referred as a series system with standbys [10, 14].

There are many application of the system, such as process and energy system, transport system, bridges, pipelines, space shuttles, etc. Reliability, availability and maintenance model of the system have been studied in the reliability literature [10, 14, 15, 16]. There are some literatures on this, but with group replacement, which is referred as opportunity based maintenance [11]. Savic et al. [13] have proved that this opportunistic problem is NP-complete, and developed genetic algorithm for analyzing this optimal opportunistic problem for real-sized systems. They analyzed different operators using a system that consists of 50 maintenance significant parts, and paid special attention to the sensitivity of solutions to the maximum number of maintenance group. A dynamic opportunistic maintenance policy for continuously monitored system has been proposed [17]. An opportunistic maintenance policy for a multi-component damage shock model with stochastically dependent components was proposed [5]. Zhaou et al. [18] introduced an opportunistic preventive maintenance (PM) scheduling algorithm based on dynamic programming for the multi-unit series system. A genetic algorithm for time and cost analysis for a Potash industry has been developed to build an intelligent maintenance system to predict whether the opportunity based maintenance strategy is cost effective or not [12].

We consider the problem as optimization of the component replacement sequence when the data is

deterministic and replacement is single. Of course, we do not consider the replacement cost or time. We show that the system is equivalent to a 'bottleneck-maximin' problem, of partitioning a set of n numbers into k parts, such that the minimum of the k part-sums is maximized. A preliminary study was carried out by Ahmed et al. [4]. It is a combinatorial optimization problem in nature. An interesting feature of combinational optimization problems is that it may be easy to hit upon the optimal solution but in contra-distinction to the 'continuous' mathematical programming problems, it is almost impossible to identify it as such by 'general methods'. No necessary conditions, let alone necessary and sufficient conditions, for a solution to be optimal, of the usual type (like the derivatives being zero) are available. However, it often happens that efficient bounds to optimal value (either globally or over well-defined subsets of the set of solutions) can be found, making possible non-trivial statements about the 'goodness' of a proposed solution. In fact, this possibility is at the heart of search algorithms like the lexisearch [2, 3, 9] and branch and bound [8] algorithms. In this paper, we are going to develop a lexisearch algorithm to find exact optimal solution and a genetic algorithm to find heuristic solution to the problem.

The paper is organized as follows: Section 2 gives a detailed statement of the problem. A lexisearch algorithm is applied in Section 3 to find exact optimal solution to the problem. Section 4 develops a genetic algorithm for the same. Section 5 describes computational experience for the algorithms. Finally, Section 6 presents comments and concluding remarks.

2. STATEMENT OF THE PROBLEM

Let $S_0 = \{C_1, C_2, \ldots\ldots, C_n\}$ be a set of n components or elements, with corresponding lives $\{t_1, t_2, \ldots\ldots, t_n\}$. A system requires k of these elements for it to be in operation. Thus initially one has k components put to work in the system, keeping the remaining n-k components as standbys, to replace 'active elements' in the system, sequentially, as and when they fail. The objective is to choose the initial set of active elements and the sequence in which the standby elements are to replace the elements on failure, so that the system life is maximized.

2.1. Illustration

Let C_i, (i =1, 2, 3, ..., 8) be 8 components, with lives $\{t_i, i=1, 2, .., 8\} = \{4, 7, 8, 9, 10, 15, 15, 20\}$ in suitable time units. Let k=3 and the replacement sequence be $\{C_1, C_2, C_3; C_4, C_5, \ldots, C_8\}$. That is, one starts with the three items C_1, C_2, C_3, with lives 4, 7 and 8 respectively, in the system, and as the active items fail, they are replaced by the items with lives 9, 10, 15, 15 and 20 in that order. The sequence of failures/ replacements can be schematically represented as shown in Table 1.

Stage No.	Cumulative time till failure	Time to failure	Active element life in respective points of failure	Lives of ordered set of standbys
0	0	0	$4(C_1)$, $7(C_2)$, $8(C_3)$	9, 10, 15, 15, 20
1	4	4	$9^*(C_4)$, $3(C_2)$, $4(C_3)$	10, 15, 15, 20
2	7	3	$6(C_4)$, $10^*(C_5)$, $1(C_1)$	15, 15, 20
3	8	1	$5(C_4)$, $9(C_5)$, $15^*(C_6)$	15, 20
4	13	5	$15^*(C_7)$, $4(C_5)$, $10(C_6)$	20
5	17	4	$11(C_7)$, $20^*(C_8)$, $6(C_6)$	*
6	23	6	$5(C_7)$, $14(C_8)$, $0(C_6)$	

TABLE 1: The sequence of failures / replacements

Starting with C_1, C_2 and C_3, we find that C_1 fails first, at time t = 4, to be replaced by C_4; with $t_4 = 9$. Now C_2, C_3 and C_4 have 3, 4 and 9 units of time left for failure and the four items C_j,(j= 5, 6, 7 and 8) with life 10, 15, 15, 20 are the ordered spares. The process is continued till the system fails totally. For the sequence (1, 2,..., 8), the system breakdown time is 23 time units. The same set of units, with different orders of replacement, leads to different system lives. For instance, the sequence (5, 1, 3, 6, 4, 8, 2, 7) gives a system life of 24 units and the sequences (4, 2, 6, 7, 8, 5, 1, 3) and (8, 1, 3, 5, 6, 7, 2, 4) lead to system lives of 26 and 27 units respectively. Our aim is to obtain the maximum system life by choosing an appropriate sequence.

2.2. Bound setting

It is easy to get some good upper bounds to the optimal values of this problem. Two such bounds are given below:

(i) Let $T_0 = \sum_{i=1}^{n} t_i$ Then, $\alpha_1 = \left\lfloor \frac{T_0}{k} \right\rfloor$ is one such upper bound.

(ii) Let the components be re-labeled, if necessary, such that $t_i \geq t_{i+1}$. Also, let $t_1, t_2 \ldots t_{r1} \geq \alpha_1$, but $t_{r1+1} < \alpha_1$. Then,

$$\alpha_2 = \frac{T_0 - \sum_{i=1}^{r_1} t_i}{k - r_1}$$

is also an upper bound. This bounding process can be recursively applied, treating the set $S_1 = S_0 - \{C_i, i = 1,2,..r_1\}$ as the set to be partitioned into $k_1 = k - r_1$ subsets, where S_0 is the set of all component lives in non-increasing order. Let us illustrate it as below.

(i) Let S_0 = {20, 15, 15, 10, 9, 8, 7, 4} and k=3. We have T_0=88, and then α_1=29. Since the sequence {20,4,8,10,15,15,9,7} leads to a solution value 27, any improvement, if at all achieved, will be not more than 2 (=29 - 27).

(ii) Let S_0= {40, 17, 10, 9, 8, 5} and k=3. Then α_1 = 29. Hence, S_1= {17, 10, 9, 8, 5} with sum, say T_1=49, and α_2 = 24 and the best partition of S_1 into two parts is equivalent to the best partition into 3 parts of S_0. The partition S_{11}={17,8}, S_{12}={10,9,5} of S_1 is of value 24 and hence a best partition of S_0 into three parts is S_{01}= {40}, S_{02}={17,8} and S_{03}={10,9,5}, with value 24.

It is worth noting that this problem, formulated as a 'bottleneck' problem in a 'dynamic background' (viz., replacing an item as and when it fails) and thus is one with 'permutations' as the 'variable' over whose domain an objective function is to be maximized, is equivalent to the 'static' problem - of partitioning a set of numbers into k parts such that the minimum among the part sums is maximized. For instance, take the case of the set {4, 7, 8, 9, 10, 15, 15, 20}, with k =3 considered already in Table 1. On examining the failure patterns table vertically with the top entries 4, 7 and 8, one is easily led to the partitions {4, 9, 15} U {7, 10, 20} U {8, 15}, with part sums 28, 37 and 23 and hence the objective function value 23 = min. {28, 37, 23}. A permutation of these parts within themselves gives rise to a different sequence but give the same objective function value. For instance, consider the following permutation: {9, 15, 4} U {7, 20, 10} U {15, 8}. The sequence is assembled as follows:

Take the first element of each of these parts, to start the system. Whichever is the smallest among them in failure time will fail first; replace it by the next element in the corresponding part. The process is repeated till an element fails but the corresponding part is 'empty'. Details are given in Table 2(a).

Failure time		Partition	Lives of standbys
Cum.	Stage	I II III	in the partitions
0	0	9 , 7, 15	{15,4}U{20,10}U{8}
7	7	2, 20*, 8	{15,4}U{10}U{8}
9	2	15*, 18, 6	{4}U{10}U{8}
15	6	9, 12, 8*	{4}U{10}U{*}
23	8	1, 4, 0	{4}U{10}U{*}

TABLE 2(a): Permutation of parts

Finally the sequence is got, in the above case, as {9, 7, 15; 20, 15, 8, 4, 10}. Obviously, if the columns I, II and III of the Table 2(a) are permuted, along with the 'part-sets', one gets a different sequence which is equivalent to the first, with the same objective function value: Thus, writing the columns as I = III, II = II, and III= I, we get the sequence {15, 7, 9; 20, 15, 8, 4, 10} as in the failure Table 2(b).

Failure time		Partition	Lives of standbys in the partitions
Cum.	Stage	I II III	
0	0	15, 7, 9	{8}U{20,10}U{15,4}
7	7	8, 20*, 2	{8}U{10}U{15,4}
9	2	6, 18, 15*	{8}U{10}U{4}
15	6	8*, 12, 9	{*}U{10}U{4}
23	8	0, 4, 1	{*}U{15}U{4}

TABLE 2(b): Permutation of parts

In fact, every partition $n=n_1+n_2+.....+n_k$ gives rise to $n_1!\ n_2!\ .....n_k!$ permutations of n components, each of which corresponds to a unique failure pattern, all of them giving the same part sums and hence the same objective function value. For instance, the same partition as above, by taking the ordering within parts as given in Table 3, leads to a different failure pattern:

{15, 4, 9} U {10, 20, 7} U {15, 8} => {15, 10, 15; 20, 4, 8, 9, {7}}

Failure time		Partition	Lives of standbys	Sequence
Cum	Stage	I II III	in the partitions	Buildup
0	0	15, 10, 15	{4, 9}U{20, 7}U{8}	*
10	10	5, 20*, 5	{4, 9} U{7} U{8}	20
15	5	4*, 15, 0	{9} U{7} U{8}	20, 4
15	0	4, 15, 8*	{9} U{7} U{*}	20, 4, 8
19	4	9*, 11, 4	{*} U{7} U{*}	20, 4, 8, 9
23	4	5, 7, 0	{*} U{7} U{*}	20, 7, 8, 9, {7}

TABLE 3: Permutation of parts

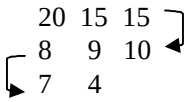
However, it should be noted that this particular permutation as shown in Table 3 leads to the situation at the final stage, where one of the 'parts' is exhausted, there are items in other parts (namely, when part III is exhausted, there is an item in part II with life 7 units), which are not yet introduced into the failure pattern table. In other words, in this example we have part III with only two elements, namely, {15, 8} has all its elements failed by 23 units; then, part I also has no spare element left, but part II has an element (of life 7 units) yet untouched as a spare. Hence, it is obvious that by transferring this element from part II to part III one can infuse another 5 (=min. {5, 7, 7}) units of life to the system. Of course, this gives rise to (or rather, changes the present partition into) the partition {15, 4, 9} U {10, 20} U {15, 8, 7} with the minimum 28 = 23 + min.{5, 7, 7}.

It is obvious that with a total $T_0 = 88$, with a part sum $S_{02} = 30$, an upper bound to the objective function is $\min\left(30, \left\lfloor \frac{88-30}{2} \right\rfloor\right) = 29$. By inspection of the present partition with S_{0i} =28, 30, 30 one can see a further, final improved partition {15, 4, 10} U {9, 20} U {7, 8, 15} leading to the best value 29.

In this particular problem, it was possible to obtain- with a little inspection and intelligent guess-work, a solution value which equaled an upper bound to the optimal solution of the problem and hence one could get a solution which could be shown to be optimal and the obtained trial solution could be recognized as optimal; which need not be the situation in general. Hence a natural question arises: is there a computationally efficient way of picking up and recognizing an optimal solution? We shall present below a procedure which appears to achieve this aim.

Without loss of generality, we shall arrange the items in a non ascending order of their values and require that the partitions S_{01}, S_{02}, ...S_{0k} are so named - or, as the algorithm proceeds, are rearranged (re-named) if necessary, such that $T_{01} \geq T_{02} \geq \ldots\ldots \geq T_{0k}$. Thus the value of the partition will be also equal to T_{0k}.

We shall illustrate the algorithm by 'working out' an example problem. Consider the same example in section 2.1. Initially we arrange the elements as {20, 15, 15, 10, 9, 8, 7, 4} and form the k = 3 subsets as per the zigzag heuristic scheme as shown below:



It is leading to the parts {20, 8, 7} U {15, 9, 4} U {15, 10}, giving the part sums 35, 28, 25 while the 'ideal' upper bound is 29. Thus, the three part sums are having as excess of +6, -1, -4 from the bound and maximum of differences between part-sums is 35-25=10. Since the part I, which is the biggest, has one element of value 7, less than 10, a simple transfer from part I to part III of this value 7 gives a better partition. But a transfer of value 7 from I to III leaves 'deficit' of 1 in I, retaining its criticality, leading to {20, 8} U {15, 9, 4} U {15, 10, 7} with part sums 28, 28, 32 with excess -1, -1, +3 and the solution value 28 . Now the one part with positive excess, namely, III does not have an element of value not more than the maximum of differences between part sums (i.e., 4) and hence, no simple transfer of items can give a better partition. Any improvement, if at all possible, could be brought about only by an exchange - and not by simple transfer of elements between parts. Since 28 is an achieved value, we can see that no optimal partition can have part sums greater than 88-(28)2= 32. If it were possible to have an improved solution, of value 29, the largest allowable part sum will be 88 - 2(29) = 30 only. Hence, one can now go for an implicit enumeration approach like the lexisearch, for picking up and establishing optimality of a partition.

As already noted, the partitions are made almost unique by defining only non-decreasing part-sums as valid. Further, the largest part sum, as just established, has to be between 30 and 29, if an optimal value of 29 can be obtained. In fact, with 29 as optimal value (the total of all parts being necessarily 88); the largest part sum should be 30 only in the present illustration. We now use the lexisearch algorithm to obtain optimal solution to the problem.

3. A LEXISEARCH ALGORITHM

The lexisearch algorithm is a systematic branch and bound approach, which derives its name from lexicography, the science of effective storage and retrieval of information. It was developed by Pandit [9], and since then it has been applied to many combinatorial optimization problems efficiently [2, 3].

In lexisearch algorithm, we first arrange the set of solutions of a problem in a hierarchy, like words in a dictionary, such that each incomplete word represents the block of words with this incomplete word as the leader of the block. We calculate bounds for the values of the objective function over these blocks of words. These bounds are then compared with the best solution value found so far. If no word in the block can be better than the best solution value found so far, then we jump over the block to the next one. However, if the bound indicates a possibility of better solutions in the block, we enter into the sub block by concatenating the present leader with appropriate letter and set a bound for the new (sub) block so obtained [2, 3, 9].

3.1. The algorithm

A pseudo-code for the lexisearch algorithm for our problem is as follows:

READ n, k and the component lives C_i, for i = 1, 2, 3, ……., n;
Set V_i = 0, for i = 1, 2, 3, ……., n;
Set X_{ij} = 0, for i = 1, 2, 3, ……., k, and j = 1, 2, 3, ……., (n-k+1);
Compute bound;

For i = 1 to (k-1) do
Set j = m = sum = index(i) = S_i = 0;
Label – j:
j = j + 1;
if(j > n) then do the following:
$S_i = S_i - C_{index(i)}$;
j = index(i);
index(i) = 0;
V_i = 0;
X_{ij} = 0;
m = m – 1;
if(m < 0) then "Optimal solution is not possible", so, Stop.
else go to Label – j;
endif;
if(V_i = 1) then
go to Label – j;
endif;
sum = $S_i + C_i$;
if(sum > bound) then
go to Label – j;
endif;
S_i = sum;
m = m + 1;
$X_{ij} = C_i$;
index(i) = j;
V_i = 1;
If(S_i = bound) then
go to Label – k;
endif;
go to Label – j;
Label – k:
endfor;
Set S_k = m = 0;
For j = 1 to n do
if(V_j = 0) then
$S_k = S_k + C_j$;
m = m + 1;
$X_{ij} = C_j$;
endif;
endfor;
Find Solution = min.{S_i, i = 1, 2, 3, ……., k};
Print the Solution and Stop.

3.2. Illustration

The 'Search Table' for the life values {20, 15, 15, 10, 9, 8, 7, 4} is shown in Table 4. Thus an optimal partition is {20, 9} U {15, 10, 4} U {15, 8, 7}. This optimal partition with sequence and failure pattern is shown in Table 5. So, the optimal sequence is {20, 15, 15; 10, 8, 9, 7, 4}.

$20_{(20)}$, $15_{(35)}$*
$15_{(35)}$*
$10_{(30)}$*
$9_{(29)}$; [15,15,10,8,7,4]
$15_{(15)}$ $15_{(30)}$*
$10_{(25)}$, $8_{(33)}$*
$7_{(32)}$*
$4_{(29)}$; [15,8,7]
$15_{(15)}$, $8_{(23)}$, $7_{(30)}$

TABLE 4: Search Table.

Failure time		Partition	Lives of standbys	Sequence
Cum	Stage	I II III	in the partitions	Buildup
0	0	20, 15, 15	{9} U {10,4} U {8,7}	*
15	15	5, 10*, 8*	{9} U {4} U {7}	10,8
20	5	9*, 5, 3	{*} U {4} U {7}	10,8,9
23	3	6, 2, 7*	{*} U {4} U {*}	10,8,9,7
25	2	4, 4*, 5	{*} U {*} U {*}	10,8,9,7,4
29	4	*, *, 1		

TABLE 5: Optimal partition with failure pattern

4. A GENETIC ALGORITHM

Genetic Algorithms (GAs) are computerized search and optimization algorithms based on the mechanics of natural genetics and natural selection [1, 7]. They are robust search algorithms which are suited for problems having comparatively larger solution spaces. However, they are essentially heuristic and, by themselves, can not guarantee the optimality of the solutions they produce.

They start from a population of chromosomes and then apply three operators: reproduction/ selection, crossover and mutation to create new, and hopefully better populations. The operator 'crossover' together with the operator 'reproduction' is the most powerful process in the GA search. The frequency of mutation is usually chosen to be considerably less than the frequency of crossover.

4.1. Genetic modeling of the problem

As a first step in applying the GA to the present problem, the solution space is to be mapped into the chromosomes of length n, the number of elements to be partitioned. In a chromosome, the genes indicate to which 'part' a particular element should belong. Then, for k-part problem of length n, one has to have k genes - e.g., 1, 2,….., k representing k-parts. The chromosome structure will then indicate the partition. For instance, let k=3 and n=8. Then, the chromosomes will be of length 8 with three genes, say, 1, 2 and 3. So, a chromosome will be the set of all ternary strings of length eight. The chromosome (3, 3, 2, 2, 1, 2, 1, 2) stands for the partition P_1= {C_5, C_7}, P_2= {C_3, C_4, C_6, C_8}, and P_3= {C_1, C_2}, where C_1, C_2,....., C_8 are the component lives arranged in non-increasing order. For any partition, the value of objective function is defined as the minimum of the part-sums. The fitness of the solution is decided by this objective function which has to be maximized. For a set of 8 components with lives {20, 15, 15, 10, 9, 8, 7, 4} arranged in non-ascending order, one of the chromosomes may be (1, 2, 3, 1, 2, 3, 1, 2) representing the partition {20, 10, 7} U {15, 9, 4} U {15, 8} with objective function value 23 = min {37, 28, 23}.

4.2. Genetic operators

There are many variations of GAs formed by using different reproduction, crossover and mutation operators. In our implementations, stochastic remainder selection method [6], the multi-point crossover operator [7] that interchanges the alternative sub-strings at randomly selected points, and the swap mutation operator [1] which selects any two genes randomly and exchanges them, have been considered.

The GA approach has been claimed to lead to very good, near-optimal solutions. However, the approach is obviously 'controlled or guided' by choice of parameters: namely, probability of crossover (P_c), probability of mutation (P_m), population size (P_s), and of course crossover points and mutation locations. As Deb [6] points out: successful working of GAs depends on a proper selection of these parameters, but often one is in the dark as to what values should be taken for these parameters. For our problem, several runs were executed with different settings of the parameters for different value

of n and k. These runs allowed us to fine-tune the parameters. After substantial testing, we settled the parameters as: P_c=0.9, P_m=0.1 and P_s=100.

5. COMPUTATIONAL EXPERIENCE

The lexisearch algorithm (LSA) and genetic algorithm (GA) have been coded in Visual C++ on a Pentium 4 personal computer with speed 3 GHz and 448 MB RAM under MS Windows XP, and tested some randomly generated problems of different sizes drawn uniformly from various ranges of data with different values of n and k. Each system contains 20 problem instances. We include two statistics to summarize the results by the algorithms: average and standard deviation of solution time and solution ratios (only for GA). Solution ratio is defined as ratio of best solution value obtained by GA to the exact solution value obtained by LSA. Tables 6 and 7 summarize the results by the algorithms for the randomly generated instances drawn from the interval [1, 100] and [1, 10000] respectively.

N	K	LSA		GA			
		Time		Sol. Ratio		Time	
		Mean	Std Dev	Mean	Std Dev	Mean	Std Dev
100	10	0.01	0.02	1.00	0.02	0.11	0.03
	20	0.03	0.05	1.00	0.05	0.15	0.05
200	20	0.13	0.07	1.00	0.03	0.22	0.09
	30	0.16	0.13	1.01	0.09	0.27	0.12
300	30	0.75	0.25	1.00	0.08	0.80	0.21
	40	0.92	0.35	1.01	0.12	0.79	0.20
400	40	2.24	1.03	1.01	0.15	2.18	0.84
	50	2.57	1.54	1.00	0.07	2.27	1.03
500	50	4.59	1.87	1.01	0.23	3.37	1.32
	60	5.64	2.43	1.02	0.56	4.77	1.55

TABLE 6: Results by the algorithms for the instances drawn from the interval [1, 100].

N	K	LSA		GA			
		Time		Sol. Ratio		Time	
		Mean	Std Dev	Mean	Std Dev	Mean	Std Dev
100	10	0.02	0.02	1.00	0.02	0.12	0.03
	20	0.03	0.03	1.00	0.02	0.15	0.05
200	20	0.14	0.08	1.01	0.05	0.23	0.07
	30	0.19	0.18	1.01	0.12	0.26	0.11
300	30	0.73	0.21	1.00	0.07	0.81	0.21
	40	0.98	0.37	1.00	0.15	0.78	0.22
400	40	2.57	0.92	1.01	0.12	2.15	0.85
	50	2.89	1.05	1.01	0.09	2.28	1.01
500	50	5.09	2.07	1.02	0.20	3.34	1.29
	60	5.78	2.73	1.02	0.65	4.76	1.51

TABLE 7: Results by the algorithms for the instances drawn from the interval [1, 10000].

It is clear from the Tables 7 and 8 that for n < 300, GA takes more time than LSA, but as size increases LSA takes more time than GA. Of course, as the size increases solution quality by GA

decreases. Also, it is seen that for same n when k increases the solution quality by GA decreases. For a value of n, computational times do not vary much for the different values of k for both algorithms.

6. DISCUSSIONS AND CONCLUSIONS

The deterministic replacement problem for serial system is viewed in such a way that lexisearch and genetic algorithms can be applied. Computational experience shows that both the algorithms are suitable for the problem. However, for the small sized instance, lexisearch algorithm is found to better one. In order to investigate the robustness of GAs, for each size, 20 instances were solved. Though the solution quality is good by GA for smaller sized instances, but the computational time is higher than that of by lexisearch algorithm. It is to be noted that for some systems the exact optimal solutions were not possible. As a whole, GA is good. The success of GA for the problem suggests the use of this technique in many other industrial fields, particularly in maintenance and reliability. Again the successful working of GA depends on the proper selection of GA parameters. So, carefully choosing the parameters may lead to a better performance of genetic algorithms on the above problem. We did not consider any case study for this problem, which may be considered, in future, to show the effectiveness of the algorithms. Also, we did not consider service cost/time, which may be considered in future.

Acknowledgements

The author wishes to acknowledge Prof. S. N. Narahari Pandit, Centre for Quantitative Methods, Osmania University, Hyderabad, India, for his valuable suggestions and moral support. The author is also thankful to the honorable anonymous reviewer for his comments and suggestions.

7. REFERENCES

[1] Z.H. Ahmed. *"Genetic algorithm for the traveling salesman problem using sequential constructive crossover".* International Journal of Biometrics & Bioinformatics 3, pp. 96-105, 2010.

[2] Z.H. Ahmed. *"A lexisearch algorithm for the bottleneck traveling salesman problem".* International Journal of Computer Science and Security 3, pp. 569-577, 2010.

[3] Z.H. Ahmed. *"A sequential constructive sampling and related approaches to combinatorial optimization".* PhD Thesis, Tezpur University, Assam, India, 2000.

[4] Z.H. Ahmed, S.N.N. Pandit, and M. Borah. *"On the solution of a deterministic replacement problem".* Presented in National Seminar on Advancing Frontiers in Statistics and Operations Research, Dibrugarh University, Assam, India, During September 22 - 24, 1997.

[5] L. Cui and H. Li. *"Opportunistic maintenance for multi-component shock models".* Journal of Mathematical Methods of Operations Research 63(3), pp. 180-191, 2006

[6] K. Deb. *"Optimization for engineering design: algorithms and examples"*. Prentice Hall of India Pvt. Ltd., New Delhi, India, 1995.

[7] D.E. Goldberg. *"Genetic algorithms in search, optimization, and machine learning"*. Addison Wesley, Reading, MA, 1989.

[8] J.D.E. Little, K.G. Murthy, D.W. Sweeny and C. Karel. *"An algorithm for the travelling salesman problem"*. Operations Research 11, pp. 972-989, 1963.

[9] S.N.N. Pandit. "*Some quantitative combinatorial search problems*". PhD Thesis, Indian Institute of Technology, Kharagpur, India, 1963.

[10] K.S. Park. *"Reliability of a system with standbys and spares".* Journal of Korean Institute of Industrial Engineering 3(1), Dec 1977.

[11] L.M. Pintelon and L.F. Gelders. *"Maintenance management decision making".* European Journal of Operational Research 58(3), pp. 209-218, 1985.

[12] M.S. Samhouri, A. Al-Ghandoor, R.H. Fouad and S.M.A. Ali. *"An intelligent opportunistic maintenance (OM) system: a genetic algorithm approach".* Jordan Journal of Mechanical and Industrial Engineering 3(4), pp. 246-251, 2009.

[13] A. Savic, G. Walters and J. Knezevic. *"Optimal, opportunistic maintenance policy using genetic algorithms, 2: analysis"*. Journal of Quality in Maintenance Engineering 1(3), pp. 25-34, 1995.

[14] K.-H. Wang and B.D. Sivazlian. *"Life cycle cost analysis for availability and reliability of series system with warm standbys components".* International Conference on Computers and Industrial Engineering, Puerto Rico, U.S.A., 1997.

[15] H. Wang. *"A survey of maintenance policies of deteriorating systems".* European Journal of Operational Research 139, pp. 469-489, 2002.

[16] H. Wang and H. Pham. *"Reliability and Optimal Maintenance".* Springer, 2006.

[17] X. Zhaou, L. Xi and J. Lee. *"A dynamic opportunistic maintenance policy for continuously monitored system".* Trans. J. of Quality Management Engineering 12(3), pp. 294-305, 2006.

[18] X. Zhaou, L. Xi and J. Lee. *"Opportunistic preventive maintenance scheduling for a multi-unit series system based on dynamic programming".* International Journal of Production Economics, 2008.