A Lexisearch Algorithm for the Bottleneck Traveling Salesman 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

The bottleneck traveling salesman problem (BTSP) is a variation of the well-known traveling salesman problem in which the objective is to minimize the maximum lap (arc length) in a tour of the salesman. In this paper, a lexisearch algorithm using adjacency representation for a tour has been developed for obtaining exact optimal solution to the problem. Then a comparative study has been carried out to show the efficiency of the algorithm as against an existing exact algorithm for some TSPLIB and randomly generated instances of different sizes.

Keywords: Bottleneck traveling salesman, Lexisearch, Bound, Alphabet table.

1. INTRODUCTION

The bottleneck traveling salesman problem (BTSP) is a variation of the benchmark traveling salesman problem (TSP). It can be defined as follows:

A network with n nodes (or cities), with 'node 1' (suppose) as 'headquarters' and a cost (or distance, or time etc.) matrix $C=[c_{ij}]$ of order n associated with ordered node pairs (i,j) is given. Let $\{1=\alpha_0, \alpha_1, \alpha_2,....,\alpha_{n-1}, \alpha_n=1\} \equiv \{1\rightarrow\alpha_1\rightarrow\alpha_2\rightarrow..... \rightarrow\alpha_{n-1}\rightarrow1\}$ be a tour, representing irreducible permutations interpreted as simple cycle. The tour value is defined as $\max \left\{ c_{\alpha_i,\alpha_{i+1}} : i = 0,1,2,...., n-1 \right\}$. The objective is to choose a tour which has minimum tour value.

Both TSP and BTSP are well known NP-hard problems. Vairaktarakis [1] considered a polynomially solvable TSP and showed that the corresponding BTSP is strongly NP-complete. The BTSP finds application in the area of workforce planning. A commonly used objective in workforce leveling (or range) is to minimize the difference between the maximum and minimum number of workers required by any worker schedule. The objective leads to level worker schedules that smooth the workforce fluctuations from one production period to the next. Such schedules are particularly useful in automobile assembly because they help to preserve overall smoothing of operations [1]. Another application of the BTSP is in minimizing makespan in a two-machine flowshop with no-wait-in-process which is a building block for more general no-wait production system [2].

Gilmore and Gomory [3] introduced the BTSP, and discussed a specific case of the problem. Definitely, the BTSP has not been as well researched as the TSP. There are a few exact algorithms available in the literature for the BTSP. An algorithm based on branch and bound (BB) is developed by Garfinkel and Gilbert [4] for solving the general BTSP, and discussed an application of the problem in the context of machine scheduling. Carpaneto et al. [5] also developed an algorithm based on BB that uses a heuristic search to find a Hamiltonian circuit containing only arcs whose cost is not greater than

the current lower bound. Ramesh [6] reported this problem as min-max TSP, and developed a lexisearch algorithm, using path representation for a tour of the salesman, to obtain exact optimal solution to the problem, and computational experiments were reported for the randomly generated problems of sizes up to 30.

There are some heuristic algorithms in the literature which are reported to be good for the general BTSP [7, 8, 9, 10]. Also, there are some algorithms in the literature which were developed for some special case of the problem [2, 11]. In this paper, we are not considering any special case of the problem, rather the general BTSP. A lexisearch algorithm, using adjacency representation for a tour of the salesman, is developed to obtain exact optimal solution to the problem. Finally, the efficiency of our algorithm is compared with the algorithm of Ramesh [6] for some TSPLIB and randomly generated instances of different sizes.

This paper is organized as follows: Section 2 presents some definitions that are required for the lexisearch algorithm. A lexisearch algorithm with an illustrative example is presented in Section 3. Computational experiments for two algorithms have been reported in Section 4. Finally, Section 5 presents comments and concluding remarks.

2. SOME DEFINITIONS

2.1. Alphabet table

Alphabet matrix, A=[a(i,j)], is a square matrix of order n formed by the positions of the elements of the cost matrix of order n, C=[c_{ij}]. The i^{th} row of the matrix A consists of the positions of the elements in the i^{th} row of the matrix C when they are arranged in the non-decreasing order of their values. If a(i,p) stands for the p^{th} element in the i^{th} row of A, then a(i,1) corresponds to the smallest element in the i^{th} row of the matrix C. That is,

$$\min_i [c_{ij}] = c_{i,a(i,1)}.\ So,\ if\ \ p < q,\ then\ \ c_{i,a(i,p)} \le c_{i,a(i,q)}.$$

Thus, the i^{th} row of A is [a(i,1), a(i,2), …., a(i,n)]. Clearly,

$$c_{i,a(i,1)} \le c_{i,a(i,2)} \le ..... \le c_{i,a(i,n)}$$

The words can be generated by considering one element in each row as follows:

$$1 \to \{a(1,j) = \alpha_1\} \to \{a(\alpha_1, k) = \alpha_2\} \to ..... \to \alpha_{n-2} \to \{a(\alpha_{n-2}, m) = \alpha_{n-1}\} \to \{\alpha_n = 1\}$$

where j, k,…, m are some indices in the alphabet matrix.

Alphabet table "$[a(i,j) - c_{i,a(i,j)}]$" is the combination of elements of matrix A and their values. For example, a cost matrix and its 'alphabet table' are shown in Table 1 and Table 2 respectively.

Node	1	2	3	4	5	6	7
1	999	75	99	9	35	63	8
2	51	999	86	46	88	29	20
3	100	5	999	16	28	35	28
4	20	45	11	999	59	53	49
5	86	63	33	65	999	76	72
6	36	53	89	31	21	999	52
7	58	31	43	67	52	60	999

TABLE 1: The cost matrix.

Node	N - V	N - V	N - V	N - V	N - V	N - V	N - V
1	7-8	4-9	5-35	6-63	2-75	3-99	1-999
2	7-20	6-29	4-46	1-51	3-86	5-88	2-999
3	2-5	4-16	5-28	7-28	6-35	1-100	3-999
4	3-11	1-20	2-45	7-49	6-53	5-59	4-999
5	3-33	2-63	4-65	7-72	6-76	1-86	5-999
6	5-21	4-31	1-36	7-52	2-53	3-89	6-999
7	2-31	3-43	5-52	1-58	6-60	4-67	7-999

TABLE 2: The alphabet table (N is the label of node, and V is the value of the node).

2.2. Incomplete word and block of words

$W = (\alpha_0, \alpha_1, \alpha_2, ...\alpha_m), m < n,$ represents an incomplete word. An incomplete word (partial tour) consists of some of the nodes. Incomplete word represents the block of words with this incomplete word as the leader of the block. If F(.) is the objective function and *W* is an incomplete word, then for a complete word *S* whose leader is *W*, we have *F(S)* ≥ *F(W).*

For the BTSP, each node is considered as a letter in an alphabet and each tour can be represented as a word with this alphabet. Thus the entire set of words in this dictionary (namely, the set of solutions) is partitioned into blocks. A block B with a leader (α_0, α_1, α_2,) of length three consists of all words beginning with (α_0, α_1, α_2,) as string of first three letters. The block A with the leader (α_0, α_1) of length 2 is the immediate superblock of B and includes B as one of its sub-blocks. The block C with leader (α_0, α_1, α_2, β) is a sub-block of block B. The block B consists of many sub-blocks (α_0, α_1, α_2, β_k), one for each β_k. The block B is the immediate super-block of block C.

By structure of the problem it is often possible to get lower bound for the block to the values of all words in a block by examining its leader. Hence, by comparing the bound with the 'best solution value' found so far, one can

(i) ‘go' into the sub-block by concatenating the present leader with an appropriate letter; if the block-bound is less than the 'best solution value',

(ii) ‘jump over’ to the next block; if no word in the block can be better in value than the 'best solution value', or

(iii) ‘jump out’ to the next super-block, if the current block, which is to be jumped over, is the last block of the present superblock.

Further, if value of the current leader is already greater than or equal to the 'best solution value' found so far, then no need for checking subsequent blocks within this super-block, and we 'jump out' to the next supper-block.

Let a, b, c, d be the four nodes in a network. The words starting with ‘a’ constitute a ‘block’ with ‘a’ as its leader. In a block, there can be many sub-blocks; for instance ‘ab’, ‘ac’ and ‘ad’ are leaders of the sub-blocks of block ‘a’. There could be blocks with only one word; for instance, the block with leader ‘abd’ has only one word ‘abdc’. All the incomplete words can be used as leaders to define blocks. For each of blocks with leader ‘ab’, ‘ac’ and ‘ad’, the block with leader ‘a’ is the immediate super-block. For example, 'go' into the sub-block for ‘db’ leads to ‘dba’ as augmented leader, 'jump over' the block for ‘abc’ is ‘abd’, and 'jump out' to the next higher order block for ‘cdbe’ is ‘cde’.

3. A LEXISEARCH ALGORITHM FOR THE BTSP

The lexicographic search derives its name from lexicography, the science of effective storage and retrieval of information. This search (lexisearch, for short) is a systematic branch and bound approach, was developed by Pandit [12], which may be summarized as follows:

The set of all possible 'solutions' to a combinatorial optimization problem is arranged in hierarchy- like words in a dictionary, such that each 'incomplete word' represents the block of words with this incomplete word as the 'leader'. Bounds are computed for the values of the objective function over these blocks of words. These are compared with the 'best solution value'. If no word in the block can be better than the 'best solution value', jump over the block to the next one. However, if the bound indicates a possibility of better solutions in the block, enter into the sub-block by concatenating the present leader with appropriate 'letter' and set a bound for the new (sub) block so obtained.

This procedure is very much like looking for a word in a dictionary; hence the name 'lexi(cographic) search'. The basic difference with the branch and bound approach is that lexisearch approach is one-pass, implicitly exhaustive, search approach, avoiding the need for book-keeping involved in storing, in active memory, the bounds, at various branching nodes at various levels and related backtracking procedures, which can be expensive in terms of memory space and computing times.

There are mainly two ways of representing salesman's path in the context of lexisearch. For example, let {1, 2, 3, 4, 5} be the labels of nodes in a 5 node instance and let path to be represented be {1→3→4→2→5→1}. Adjacency representation of this path is usual representation of corresponding permutation, namely, $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 4 & 2 & 1 \end{pmatrix}$, indicating that the edges 1→3, 2→5, …., 5→1 constitute the tour. The path representation just lists the sequence of the tour as (1, 3, 4, 2, 5). The following subsections discuss the lexisearch algorithm by considering adjacency representation for solving the BTSP and its illustration through an example.

3.1. The algorithm

Ramesh [6] used path representation for a tour to obtain exact optimal solution to the problem. As reported, the algorithm shows a large variation in solution times. So, we present another lexisearch algorithm using adjacency representation for a tour. A preliminary version of this algorithm is reported in Ahmed [8]. The algorithm is presented below:

Let $C=[c_{ij}]$ be the given n x n cost matrix and c_{ij} be the cost of visiting of node j from node i, and let 'node 1' be the starting node.

Step 0: - Form the 'alphabet table'. Initialize the 'best solution value' to a large number, and set l = 1.

Step 1: - With the partial tour of length (l -1) take as leader; consider the first 'legitimate and unchecked' node. Compute the lower bound as discussed in section 3.2, and go to step 2. If there is no any 'legitimate and unchecked' node, go to step 5.

Step 2: - If the lower bound is less than the 'best solution value', go to step 3, else go to step 5.

Step 3: - If there is a sub-tour, go to step 1, else go to step 4.

Step 4: - Go to sub-block, i.e., augment the current leader; concatenate the considered node to it, lengthening the leader by one node, and compute the current tour value. If the current tour is a complete tour, then replace the 'best solution value' with the current solution value, and go to step 5. If the current tour is not a complete tour, then go to step 1.

Step 5: - Jump this block, i.e., decrement l by 1 (one), rejecting all the subsequent tours from this block. If l<1, go to step 6, else go to step 1.

Step 6: - Current tour gives the optimal tour sequence, with 'best solution value' as the optimal cost, and stop.

3.2. Lower bound

The objective of lower bound is to skip as many subproblems in the search procedure as possible. A subproblem is skipped if its lower bound exceeds the 'best solution value' found so far in the process. The higher the lower bound the larger the set of subproblems that are skipped. Some algorithms in the literature calculate overall lower bound for the BTSP instance and develop algorithms based on relaxation and subtour elimination scheme [4, 5, 9]. Our lexisearch algorithms do not follow this way for solving the BTSP instances. In our algorithm, we are not setting lower bound for an instance, rather setting lower bound for each leader on the value of objective function for the instance as follows:

Suppose the present permutation for the partial tour is $\begin{pmatrix} 1 & 2 & 3 \\ \alpha_1 & \alpha_2 & \alpha_3 \end{pmatrix}$ and the node α_4 is selected for concatenation. Before concatenation, we check the bound for the leader $\begin{pmatrix} 1 & 2 & 3 & 4 \\ \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 \end{pmatrix}$. For that, we start our computation from 5th row of the 'alphabet table' and traverse up to the nth row, check the value of first 'legitimate' node (the node that is not present in the partial tour) in each row. Maximum among the values of first 'legitimate' nodes and the leader value is the lower bound for the leader $\begin{pmatrix} 1 & 2 & 3 & 4 \\ \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 \end{pmatrix}$.

3.3. Illustration

Working of the above algorithm is explained through an example of the seven-node instance given in Table-1. Table 3 gives the 'search table'. The symbols used therein are listed below:

GS: Go into the sub-block, i.e., attach the first 'free' letter to the current leader.

JB: Jump over the block, i.e., go to the next block of the same order i.e., replace the last letter of the current block by the letter next to it in the alphabet table.

JO: Jump out to the next, higher order block, i.e., drop out the last letter of the current leader and then jump the block.

BS: Best solution value.

ST: Sub-tour.

As illustration of the example, we consider BS = 9999 and 'partial tour value (Sol)' = 0. We start from 1st row of the 'alphabet table'. Here, a(1,1) = 7 with 'present node value (Val)' = c_{17} = 8. Since Max{Sol, Val} = 8 < BS, we go for bound calculation for the present leader $\begin{pmatrix} 1 \\ 7 \end{pmatrix}$. The bound will guide us whether the node 7 will be accepted or not.

$$\begin{aligned} Bound &= Max\ \{Sol\ , Val\ , c_{2,a(2,4)}, c_{3,a(3,1)}, c_{4,a(4,1)}, c_{5,a(5,1)}, c_{6,a(6,1)}, c_{7,a(7,1)}\} \\ &= Max\ \{0, 8, c_{2,6}, c_{3,2}, c_{4,3}, c_{5,3}, c_{6,5}, c_{7,2}\} \\ &= Max\ \{0, 8, 29, 5, 11, 33, 21, 31\} = 33 \end{aligned}$$

Since Bound<BS, we accept the node 7 that leads to the partial permutation $\begin{pmatrix} 1 \\ 7 \end{pmatrix}$ with Sol=8. Next we go to 2th row of the 'alphabet table'. Since a(2,1) = 7 is repeated, we consider the next element of the row, i.e., a(2,2) =6 with Val = c_{26} = 29. Since Max{Sol, Val} = 29 < BS, we go for bound calculation for the present leader $\begin{pmatrix} 1 & 2 \\ 7 & 6 \end{pmatrix}$.

Leaders							Bound	Best Solution Value	Remarks
1	2	3	4	5	6	7			

7-8						33	9999	GS
	6-29					33	9999	GS
		2-5				33	9999	GS
			3-11			65	9999	GS
				4-65		65	9999	GS
					5-21	65	9999	ST
					1-36	65	9999	GS
						5-52	65	9999	GS
						BS =	65	JB, JO
			1-20			43	65	GS
				3-33		52	65	GS
					5-21	67	65	ST
					4-31	52	65	GS
						5-52	52	65	GS
						BS =	52	JB, JO
			5-59			59	52	JO
		4-16				33	52	GS
			3-11			63	52	ST
			1-20			33	52	GS
				3-33		33	52	GS
					5-21	33	52	GS
						2-31	33	52	GS
						BS =	33	JB, JO
			2-45			45	33	JO
		5-28				33	33	JB
		6-35				35	33	JO
	4-46					46	33	JO
4-9						33	33	JB
5-35						35	33	STOP

TABLE 3: The search table.

$$Bound = Max\{Sol, Val, c_{3,a(3,1)}, c_{4,a(4,1)}, c_{5,a(5,1)}, c_{6,a(6,1)}, c_{7,a(7,1)}\}$$
$$= Max\{8, 29, c_{3,2}, c_{4,3}, c_{5,3}, c_{6,5}, c_{7,2}\}$$
$$= Max\{8, 29, 5, 11, 33, 21, 31\} = 33$$

Since Bound < BS, we accept the node 6 that leads to the partial permutation $\begin{pmatrix} 1 & 2 \\ 7 & 6 \end{pmatrix}$ with Sol=29. Proceeding in this way, we obtain the 1st complete permutation $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 7 & 6 & 2 & 3 & 4 & 1 & 5 \end{pmatrix}$ for the tour {1→7→5→4→3→2→6→1} with Sol= 65. Since Sol<BS, so we replace BS = 65. Now, we jump out to the next higher order block, i.e., $\begin{pmatrix} 1 & 2 & 3 \\ 7 & 6 & 2 \end{pmatrix}$ with Sol = 29, and try to compute another complete tour with lesser tour value. Proceeding in this way, we obtain the optimal tour {1→7→2→6→5→3→4→1} that is given by the permutation $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 7 & 6 & 4 & 1 & 3 & 5 & 2 \end{pmatrix}$ with optimal solution value = 33.

4. COMPUTATIONAL EXPERIMENT

Our lexisearch algorithm (LSA) has been encoded in Visual C++ on a Pentium 4 personal computer with speed 3 GHz and 448 MB RAM under MS Windows XP. Also, for the comparison lexisearch algorithm by Ramesh [6], named as RA, is encoded and run in the same environment. Both the algorithms (RA and LSA) are tested for some TSPLIB instances and randomly generated instances of different sizes drawn from different uniform distribution of integers.

Table 4 gives the results for nine asymmetric TSPLIB instances of size from 17 to 70. We report optimal solution values and solution times (in second) for solving the instances by both RA and LSA. To the best of our knowledge, no literature presents experimental solutions to the asymmetric TSPLIB instances. The instance br17 of size 17 could be solved within only 1.59 seconds by RA, which could

be solved by LSA within 185.99 seconds. For the remaining instances, reported in the table, LSA is found to be better. We do not report the solution of other asymmetric TSPLIB instances, because we could not solve them by any algorithm within one hour. Table 4 also reports the computational time when the optimal solution is seen for the first time. In fact, a lexisearch algorithm first finds an optimal solution and then proves the optimality of that solution, i.e., all the remaining subproblems are discarded. Table 4 shows that, on average solution time, RA found optimal solution within 41% of the total solution time, whereas LSA found the optimal solution within only 6% of the total solution time. That is, RA spent 59% and LSA spent 94% of total time on proving optimality of the solutions. Therefore, for these asymmetric TSPLIB instances, RA spends a relatively large amount of time on finding an optimal solution compared to our LSA, and hence, a small number of subproblems are thrown by RA.

Instances	n	Optimal Solution	Solution time		Solution is seen first	
			RA	LSA	RA	LSA
br17	17	8	1.59	185.99	0.39	0.00
ftv33	34	113	0.16	1.32	0.16	0.55
ftv35	36	113	0.34	0.48	0.34	0.42
ftv38	39	113	0.02	0.05	0.02	0.05
p43	43	5008	0.02	0.02	0.05	0.02
ftv44	45	113	57.88	49.13	57.88	40.23
ft53	53	977	2960.56	1970.3	0.00	0.00
ftv64	65	104	812.03	0.00	789.23	0.00
ft70	70	1398	1452.36	156.45	1278.93	94.32
Mean			**587.22**	**262.64**	**236.33**	**15.07**

TABLE 4: Solution times for asymmetric TSPLIB instances

Instances	n	Optimal Solution	Solution time		Solution is seen first	
			RA	LSA	RA	LSA
burma14	14	418	0.08	0.03	0.01	0.00
ulysses16	16	1504	15.21	0.03	0.00	0.00
gr17	17	282	105.36	75.98	0.00	0.02
gr21	21	355	293.21	315.06	0.00	0.02
ulysses22	22	1504	325.09	270.13	0.00	0.00
gr24	24	108	121.56	29.08	78.88	0.03
fri26	26	93	87.98	70.32	14.05	8.94
bayg29	29	111	2972.05	108.06	2089.32	0.01
bays29	29	154	2145.32	75.32	0.00	0.00
swiss42	42	67	**----**	2148.5	-----	40.17
bier127	127	7486	**----**	3541.2	0.03	0.01
Mean			**673.98**	**603.06**	**218.23**	**4.47**

TABLE 5: Solution times for symmetric TSPLIB instances

Table 5 gives the results for eleven symmetric TSPLIB instances of size from 14 to 127. Recently, Ramakrishnan et al. [9] developed a very good heuristic algorithm for the general BTSP and reported results for some symmetric TSPLIB instances only. Since, the nature of our algorithm is not same as their algorithm; we do not to carry out comparison with the algorithm in terms of solution times. However, solutions reported there are found to be same as our solutions. Out of eleven instances two instances could not be solved within one hour by RA. Of course, we saw the optimal solution for one of them within 0.03 seconds. On the basis of average solution times, the table concludes that LSA is

better than RA. It is to be noted that while we calculate average solution time, we do not consider the instances which were not solved optimally within one hour. For these symmetric TSPLIB instances also, RA spends a relatively large amount of time on finding an optimal solution compared to LSA. For these case also, we do not report the instances which could not be solved by any algorithm within one hour.

Randomly generated asymmetric and symmetric instances of different sizes are drawn from uniform distribution of integers in the intervals [1, 100] and [1, 10000]. Fifteen different instances were generated for each size. Table 6 reports mean and standard deviation of solution times by RA and LSA for asymmetric instances. On the basis of the average solution times and standard deviation, Table 6 shows that LSA is better than RA for both intervals. Of course, instances generated from the interval [1, 10000] are found to be more difficult than the instances generated from [1, 100].

We report mean and standard deviation of solution times by RA and LSA for symmetric instances in the Table 7. For these instances also, LSA is found to be better than RA. Also, the instances generated from [1, 10000] are found to be more difficult than the instances generated from [1, 100]. It is also observed that symmetric instances are more difficult than the asymmetric instances.

n	$1 \le c_{ij} \le 100$				$1 \le c_{ij} \le 10000$			
	RA		LSA		RA		LSA	
	Mean	Std dev	Mean	Std dev	Mean	Std dev	Mean	Std dev
30	24.58	39.33	10.12	25.85	27.18	40.32	13.25	29.18
35	141.93	198.96	65.06	119.38	158.21	201.15	75.06	146.16
40	457.95	578.08	163.28	319.3	495.92	518.15	196.86	347.01
45	735.14	952.32	275.03	502.32	802.21	1103.87	289.50	562.23
50	959.21	1014.23	317.92	516.39	967.32	1201.14	321.15	596.27

TABLE 6: Solution times for random asymmetric instances.

n	$1 \le c_{ij} \le 100$				$1 \le c_{ij} \le 10000$			
	RA		LSA		RA		LSA	
	Mean	Std dev	Mean	Std dev	Mean	Std dev	Mean	Std dev
30	29.58	43.12	15.22	20.15	32.16	47.52	14.99	21.35
35	189.21	209.56	76.98	101.76	190.46	229.13	79.32	98.21
40	507.19	618.54	182.35	357.97	535.78	761.76	166.28	375.21
45	805.98	987.32	352.67	547.12	854.36	1087.25	398.52	601.24
50	1020.01	1321.45	573.87	602.95	1223.01	1532.78	1052.32	1524.21

TABLE 7: Solution times for random symmetric instances.

5. CONCLUSION & FUTURE WORK

We presented a lexisearch algorithm using adjacency representation method for a tour for the bottleneck traveling salesman problem to obtain exact optimal solution to the problem. The performance of our algorithm is compared with the lexisearch algorithm of Ramesh [6] for some TSPLIB instances and two types of randomly generated instances of different sizes. The computational experiment shows that our lexisearch algorithm is better. Between asymmetric and symmetric TSPLIB as well as random instances, symmetric instances are found to be hard.

In this present study, it is very difficult to say that what moderate sized instance is unsolvable by our lexisearch algorithm, because, for example, br17 of size 17 could be solved within 185.99 seconds

and dantzig42 of size 42 could not be solved within one hour, whereas, ftv64 of size 65 could be solved within only 0.00 seconds by our algorithm. It certainly depends upon the structure of the instance. So a closer look at the structure of the instances and then developing a data-guided module may further reduce the solution time. Also, it is seen that the optimal solution is seen for the first time very quickly, which suggests that applying a tight lower bound method may reduce the solution time drastically, which are under our investigations.

Acknowledgements

The author wishes to acknowledge Prof. S.N. Narahari Pandit, Hyderabad, India for his valuable suggestions and moral support. The author also thanks the reviewer for his constructive suggestions.

6. REFERENCES

[1] G.L. Vairaktarakis. "*On Gilmore-Gomory's open question for the bottleneck TSP*". Operations Research Letters 31, pp. 483–491, 2003.

[2] M.-Y. Kao and M. Sanghi. *"An approximation algorithm for a bottleneck traveling salesman problem".* Journal of Discrete Algorithms, doi: 10.1016/j.jda.2008.11.007 (2009)

[3] P.C. Gilmore and R.E. Gomory. *"Sequencing a one state-variable machine: a solvable case of the traveling salesman problem"*. Operations Research 12, pp. 655–679, 1964.

[4] R.S. Garfinkel and K.C. Gilbert. *"The bottleneck traveling salesman problem: Algorithms and probabilistic analysis"*. Journal of ACM 25, pp. 435-448, 1978.

[5] G. Carpaneto, S. Martello and P. Toth. *"An algorithm for the bottleneck traveling salesman problems".* Operations Research 2, pp. 380-389, 1984.

[6] M. Ramesh. *"A lexisearch approach to some combinatorial programming problems".* PhD Thesis, University of Hyderabad, India, 1997.

[7] Z.H. Ahmed, S.N.N. Pandit, and M. Borah. *"Genetic Algorithms for the Min-Max Travelling Salesman Problem".* Proceedings of the Annual Technical Session, Assam Science Society, India, pp. 64-71, 1999.

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

[9] R. Ramakrishnan, P. Sharma and A.P. Punnen. *"An efficient heuristic algorithm for the bottleneck traveling salesman problem".* Opsearch 46(3), pp.275-288, 2009.

[10] J. Larusic, A.P. Punnen and E. Aubanel. *"Experimental analysis of heuristics for the bottleneck traveling salesman problem".* Technical report, Simon Fraster University, Canada, 2010.

[11] J.M. Phillips, A.P. Punnen and S.N. Kabadi. "*A linear time algorithm for the bottleneck traveling salesman problem on a Halin graph*". Information Processing Letters 67, pp. 105-110, 1998.

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

[13] TSPLIB, http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95/