A New Data Structure for Lin-Kernighan Traveling Salesman Heuristic

Hung Dinh NGUYEN[1], Ikuo YOSHIHARA[2], Kunihito YAMAMORI[3], Moritoshi YASUNAGA[4]

Abstract

The Lin-Kernighan (LK) heuristic is one of the most effective and efficient algorithms for the Traveling Salesman Problem (TSP). However, the LK heuristic is quite complicated and has many choices for implementing it. Especially, the data structure for tour representation plays an important role in the LK's performance. Traditionally, binary trees (including splay trees) are asymptotically the best tour representation. Empirically, however, they are utilized only for solving problems with more than one million cities due to the large overhead. Arrays are suitable for solving problems having up to a thousand cities and two-level trees are used for the problems with a thousand to a million cities. This paper proposes three-level trees as a new data structure. Although this structure is asymptotically not better than the existing ones, it perform empirically better than the existing ones in the range being investigated in this study (from 10^3 to $10^{6.5}$ cities).

Key Words:
Traveling salesman problem, Combinatorial optimization, Lin-Kernighan heuristic, Data structure, Algorithm analysis, Three-level trees

1. Introduction

The Traveling Salesman Problem (TSP) is one of the most important and representative combinatorial optimization problems, because it is simple to state and widely applicable but difficult to solve. The TSP has applications in many fields such as vehicle routing, robot control, and crystallography. For example, the problems of collecting coins in automatic vending machines, scheduling jobs in a single machine, and ordering drill holes in a circuit board can all be formulated as TSPs.

The TSP can be stated as follows. In the TSP, locations of all the cities are given and the salesman's task is to find the cheapest route connecting them all, with each city visited only once and return to the city of origin. The cost here can be distance, time, or money etc. This paper only deals with symmetric TSPs, in which all the costs between any two cities are equal in both directions.

1) Graduate Student, Graduate School of Engineering, Miyazaki University

2) Professor, Dept. of Computer Science and Systems Engineering, Miyazaki University

3) Associate Professor, Dept. of Computer Science and Systems Engineering, Miyazaki University

4) Associate Professor, Institute of Information Sciences and Electronics, University of Tsukuba

The TSP is NP-complete. Any method that guarantees to find the true optimal solution requires running time growing exponentially with the size of problem. The biggest TSP solved to optimality so far has 15,112 cities. The problem has been solved by a cutting-plane algorithm[1], running on a network of computers. The equivalent sequential time is about 22.6 years of a 500-MHz Compaq computer. Therefore, to attack larger problems, we need to develop approximate algorithms that do not aim at finding the true optimal solution but at finding a good solution in an acceptable running time. Among approximate algorithms for the TSP, the Lin-Kernighan (LK) heuristic[2] is one of the most effective algorithms. Since the LK algorithm was proposed, it had been the champion heuristic for the TSP for about 15 years. Furthermore, most of algorithms that supersede the LK algorithm are meta-heuristics including it in one or other ways, e.g., chained LK[3),4)], iterated LK[5], and hybrid Genetic Algorithms[6)-9)]. Thus, improvements to the LK heuristic are always of significant, since they improve not only the algorithm itself but also other algorithms that include it.

The LK heuristic is very complicated and has many choices for implementing it. Especially, the data structure for tour representation plays an important role in the LK's performance. Traditionally, arrays have been used for tour representation for problems with the size up to a thousand cities. Two-level trees have been used for problems with a thousand to a million cities and binary trees (including splay trees) have been used for problems with more than a million cities.

This paper newly proposes three-level trees for tour representation for large problems. Although the new data structure is asymptotically not better than the conventional ones, it performs empirically better than the conventional ones in the range covered by this study (from 10^3 to $10^{6.5}$ cities). Furthermore, the splay tree representation, which has the best time complexity, does not seem to catch up with it until the number of cities reaches 10^7.

The paper is organized as follows. Section 2 presents the conventional data structures. Section 3 describes the new data structure. Experiments and validations of the new data structure are presented in section 4. Finally, conclusions are given in Section 5.

2. Conventional Data Structures

In the abstract level, a data structure for representing a tour in the LK heuristic must support the following four basic operations:

- *Next(T, a)*: This operation returns the successor of *a* in tour *T*.
- *Prev(T, a)*: This operation returns the predecessor of *a* in tour *T*.
- *Between(T, a, b, c)*: This operation returns true if *b* lies between *a* and *c* in tour *T* (suppose that the direction for traversing the tour is well defined). It returns false otherwise.
- *Flip(T, a, b)*: This operation reverses the segment between *a* and *b* in tour *T*.

Which data structure should be used for tour representation depends mainly on the size of the problems to be solved. Here we will briefly discuss three conventional tour representations: arrays, two-level trees, splay trees, which are most commonly used in current implementations of the LK heuristic. More details about these tour representations can be found in Fredman *et al.*[10].

2.1 Arrays

The array data structure is the most straightforward implementation of the tour representation used in the LK algorithm. The tour is represented by two one-dimensional arrays of length *N*. (Throughout this paper, *N* is used to denote the number of cities.) The first array contains the names of the cities in visiting order. The second array contains the indices of the cities. It is an inversion of the first array. For example, if city *i* is to be visited at the *j*th visit, then the value at the *j*th position of the first array is *i*, and the value at the *i*th position of the second array is *j*. With these two arrays, the three queries *Next*, *Prev*, and *Between* can be implemented in *O(1)* amortized time. The *Flip* operation takes an amortized worst-case time *O(N)*. Therefore, this representation is not well scalable with the size of problem. Given its small constant time overhead for the three query operations, however, the array representation is appropriate for solving problems with up to a thousand of cities.

2.2 Two-level Trees

The two-level tree data structure was first proposed for tour representation by Chrobak *et al.*[11]. The idea is to divide the tour into roughly $N^{1/2}$ segments, each of which having length in the range from $N^{1/2}/2$ to $2N^{1/2}$ cities. These segments are maintained as a doubly-linked list of nodes. It has been proved that the amortized times for performing the *Next*, *Prev*, and *Between* operations of the two-level tree representation are all *O(1)* while the amortized worst-case time for performing the *Flip* operation is $O(N^{1/2})$ if we always rebalance the tree after each *Flip*, or *O(N)* if we do not rebalance it. This is currently the fastest and most robust tour representation for solving problems ranging in size from a thousand to a million of cities. Many real-world TSP applications have sizes falling in this range.

2.3 Splay Trees

Another way for representing a tour is using binary search trees, with each node represents a vertex in the tour and each subtree represents a subtour. To facilitate the *Flip* operation, each node is attached with a reversal bit that indicates in what order the subtree rooted at that node should be traversed. If the reversal bit of a node is off (on), the subtree rooted at that node should be traversed in in-order (reversed in-order). In the case that all of the four tour operations are performed by using the *Splay* operation[10], binary trees are also called splay trees. It has been proved that all of the four tour operations of splay trees can be implemented in *O(logN)* worst-case time if we always rebalance the trees after each *Flip*. This is the best time complexity that has been known for tour representations used in the LK heuristic. However, due to the large overhead, splay trees are only suitable for solving very large problems (a million or more cities).

3. New Data Structure: Three-level Trees

We analyzed that although two-level trees need to be kept roughly balance to yield the $O(N^{1/2})$ amortized time for the *Flip* operation in the worst case, most of current LK implementations do not implement it in practice. Without rebalancing, the amortized time for doing the *Flip* operation of two-level trees in the worst case is *O(N)*, the same as with the array data structure. The performance of two-level trees without rebalancing, however, has been proved to be much better than the array data structure when solving large problems. It is probably because the amortized average-case time for doing the *Flip* operation of two-level trees without rebalancing is better than that of the array data structure.

Based on this analysis, we propose three-level trees to represent the TSP tour for solving large problems. The three-level tree data structure, as its name inspires, has 3 levels. The first (children) level has N elements. The second (parent) level has roughly $N^{2/3}$ elements and the third (grandparent) level has roughly $N^{1/3}$ elements. Similar to two-level trees, the elements of each level of three-level trees are maintained in a doubly-linked list. It can be easily seen that three operations *Next*, *Prev*, and *Between* of three-level trees take amortized time *O(1)*, the same as with two-level trees, but with larger overheads. The amortized time for performing the *Flip* operation of three-level trees in the worst case is $O(N^{2/3})$ with rebalancing (since there are roughly $N^{2/3}$ elements at the second level), or *O(N)* without rebalancing.

In the case that rebalancing is not implemented, both two-level tree and three-level tree data structures have the same amortized worst-case time *O(N)* for doing the *Flip* operation. However, we expect that in practice the performance of three-level trees is better than that of two-level trees when solving large problems.

It is quite straightforward to extend two-level trees to three-level trees, so we will not present the detail implementations of our three-level trees here. Instead, the readers are recommended to refer to the two-level tree data structure in Fredman *et al.*[10].

4. Experiments and Validations

We must first mention about our LK implementation. Our code was written in the C programming language. To save time, we only coded the LK search engine and the three-level trees. Other parts such as reading problems, building *k*-d trees, creating neighbor lists, generating initial tours etc. were taken from the Chained Lin-Kernighan heuristic[2], which is a part of the *Concorde* optimization code that has been written by Applegate *et al.*[1], and made available on their TSP homepage (http://www.math.princeton.edu/tsp/) for research purposes. The code for the array, two-level tree, and splay tree representations was also taken from the above source.

TSP benchmarks are taken from the TSPLIB[12] and the 8th DIMACS challenge homepage on the TSP (http://www.research.att.com/~dsj/chtsp/). They are a part of the testbed that Johnson and McGeoch[13] have used for studying the asymptotic behavior of TSP heuristics. The benchmarks are divided into four groups as follows:

(1) Random uniform Euclidean group: This group includes 25 problems, ranging in size from 10^3 to $10^{6.5}$ cities. There are 10 problems with $N = 10^3$, 5 problems with $N = 10^{3.5}$, 3 problems with $N = 10^4$, 2 problems with each size $N = 10^{4.5}$ and $N = 10^5$, and one problem with each of the size $N = 10^{5.5}$, $N = 10^6$, and $N = 10^{6.5}$.

(2) Random clustered Euclidean group: This group includes 23 problems, ranging in size from 10^3 to $10^{5.5}$ cities. The numbers of problems for each size are the same with those in the first group (for sizes from 10^3 to $10^{5.5}$).

(3) Random matrix group: This group includes 7 problems, ranging in size from 10^3 to 10^4 cities. There are 4

problems with $N = 10^3$, 2 problems with size $N = 10^{3.5}$, and one problem with $N = 10^4$.

(4) TSPLIB group: This group includes 11 symmetric problems from the TSPLIB. For the entry $N = 10^3$, four problems pr1002, pcb1173, rl1304, and nrw1379 are used. For the entry $N = 10^{3.5}$, three problems pr2392, pcb3038, and fnl4461 are used. For the entry $N = 10^4$, two problems pla7397 and brd14051 are used. The problems pla33810 and pla85900 are used for entries $N = 10^{4.5}$ and $N = 10^5$, respectively.

Since the performance of the LK algorithm for various tour representations depends on machines, we used two machines for our experiments. The first machine is a 1.6 GHz Pentium IV (Pentium hereafter) and the second one is a Sun Ultra 80 (Ultra hereafter). Both machines have 1 GB of main memory. The Pentium is running under Windows 2000 and the Ultra is running under Solaris 2.6. We compiled our codes using the MSC++ 6.0 on the Pentium and the GNU **gcc** version 2.95.2 with its '-O3' optimization option on the Ultra.

Experimental results on the Pentium and Ultra machines are given in Table 1 and Table 2, respectively.

In these tables, ARY, 2LT, SPT, and 3LT denote the array, two-level tree, splay tree, and three-level tree representations, respectively. Since the preprocessing time dominates a large portion of the total running time, we measured it and the running time of the LK optimization part separately to see more clearly the differences.

The first thing that can be said from the table is that ARY is the worst tour representation in the range being investigated. On the Pentium machine, ARY wins SPT only for the 10^3-city clustered problems. It should be note, however, that the ARY representation suffers from a phenomenon, which has already been observed by Fredman *et al.*[10], that on certain types of machine, their performance for the *Flip* operation degrades rapidly as N gets large. This phenomenon appears more effectively on the Pentium machine than on the Ultra machine. For example, the Pentium is approximately two times faster than the Ultra when solving 10^3-city uniform problems. However, it is only 1.3 times faster when $N = 10^5$ and by the time $N = 10^{5.5}$, the Ultra is even slightly faster than the Pentium. Therefore, the results of the ARY representation probably depend greatly on machines.

The second observation is that 2LT wins SPT for all but the two biggest cases ($N = 10^6$ and $N = 10^{6.5}$) of random uniform Euclidean problems. Fredman *et al.*[10] reported slightly different results for their LK implementation, with 2LT still wins SPT for the case $N = 10^6$ of uniform problems.

The third observation is the good performance of the 3LT representation. To our surprise, it starts to win 2LT by the time $N = 10^4$ for uniform problems, $N = 10^{4.5}$ for clustered and TSPLIB problems, and $N = 10^{3.5}$ for matrix problems. 3LT is nearly 1.6 times faster than 2LT when solving the $10^{6.5}$-city uniform problem. (Throughout this paper, speedups for the LK heuristic are calculated without including the preprocessing times.) The Ultra machine also gave similar results, although the crossover points are slightly different. 3LT wins 2LT by the time $N = 10^{3.5}$ for uniform problems, $N = 10^3$ for matrix problems, and $N = 10^5$ for TSPLIB problems.

Since 3LT has slower *Next*, *Prev*, and *Between* operations compared to 2LT, its *Flip* operation must be faster. This supports our hypothesis that 3LT has better amortized average-case time for doing the *Flip* operation than 2LT.

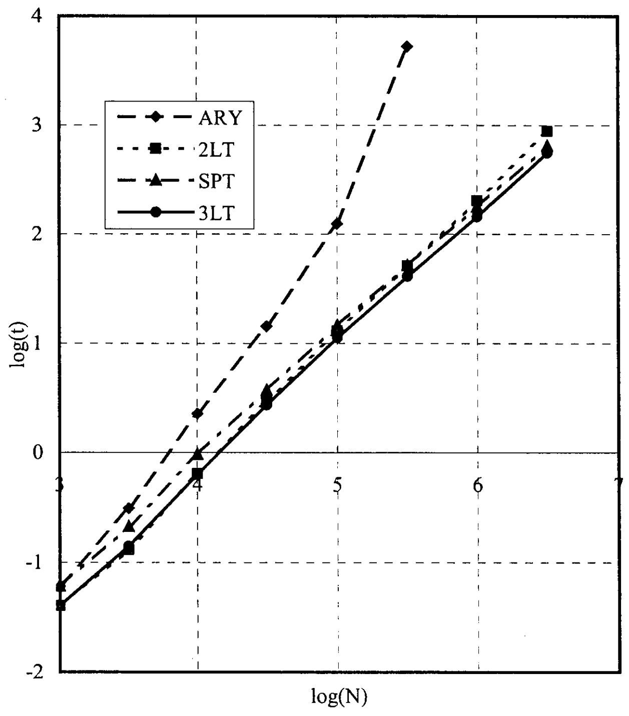

Fig. 1: Time growth rates of various tour representations on random uniform Euclidean problems (Pentium).

Table 1: LK Time (Seconds) on a Pentium IV 1.6 GHz for Various Tour Representations

N =	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6	$10^{6.5}$
Random Uniform Euclidean Problems								
Preprocessing	0.04	0.12	0.43	1.75	8.06	30.40	102.52	359.28
ARY	0.06	0.31	2.26	14.25	123.96	5252.92	–	–
2LT	0.04	0.13	0.64	2.92	12.85	50.84	203.45	876.53
SPT	0.06	0.21	0.96	3.69	14.53	51.50	177.53	640.86
3LT	0.04	0.14	0.63	2.71	11.17	40.84	144.36	556.14
Random Clustered Euclidean Problems								
Preprocessing	0.04	0.13	0.47	2.05	10.02	37.24		
ARY	0.18	0.70	2.47	10.51	54.22	577.61		
2LT	0.14	0.41	1.56	6.04	24.40	96.36		
SPT	0.21	0.64	2.37	8.09	29.61	105.67		
3LT	0.15	0.43	1.60	5.91	22.93	85.38		
Random Matrix Problems								
Preprocessing	0.32	3.25	44.72					
ARY	0.32	3.60	42.86					
2LT	0.07	0.40	3.09					
SPT	0.12	0.52	4.13					
3LT	0.07	0.36	2.55					
TSPLIB Problems								
Preprocessing	0.04	0.11	0.41	1.63	4.17			
ARY	0.07	0.34	3.11	10.83	44.38			
2LT	0.05	0.16	0.78	3.42	8.50			
SPT	0.07	0.22	1.03	4.05	9.94			
3LT	0.06	0.16	0.78	3.37	8.31			

Table 2: LK Time (Seconds) on a SUN Ultra 80 for Various Tour Representations

N =	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6	$10^{6.5}$
Random Uniform Euclidean Problems								
Preprocessing	0.07	0.24	0.83	2.97	13.74	56.91	205.20	764.58
ARY	0.13	0.54	3.32	21.54	166.44	4772.22	–	–
2LT	0.11	0.38	1.35	6.16	27.49	118.02	482.10	2031.78
SPT	0.16	0.49	1.97	8.50	34.41	126.25	450.92	1597.59
3LT	0.12	0.36	1.33	5.71	24.58	93.18	347.42	1312.02
Random Clustered Euclidean Problems								
Preprocessing	0.07	0.27	1.01	3.47	16.72	70.26		
ARY	0.45	1.50	4.76	19.54	100.89	814.56		
2LT	0.40	1.20	3.83	15.76	69.16	278.75		
SPT	0.55	1.72	5.39	21.54	83.84	289.61		
3LT	0.43	1.25	3.92	15.06	64.21	243.16		
Random Matrix Problems								
Preprocessing	1.08	11.57	125.36					
ARY	0.47	4.91	59.94					
2LT	0.18	0.84	5.56					
SPT	0.26	1.19	6.81					
3LT	0.17	0.81	4.34					
TSPLIB Problems								
Preprocessing	0.08	0.24	0.85	2.56	6.94			
ARY	0.17	0.71	4.27	23.47	94.01			
2LT	0.14	0.42	1.67	7.34	16.30			
SPT	0.19	0.56	2.13	8.59	18.71			
3LT	0.15	0.44	1.69	7.41	16.26			

On both machines, 3LT is always faster than SPT in the size range covered by this study. Although appearing to have slightly worse observed growth rate than SPT (Fig. 1), 3LT is still approximately 1.2 times faster than SPT when solving the $10^{6.5}$-city uniform problem. Thus, 3LT is likely comparable with SPT by the time $N = 10^7$.

4. Conclusions

In this paper, three-level trees have been proposed as a new data structure for representing tour used in the LK heuristic. Benchmarks ranging in sizes from 10^3 to $10^{6.5}$ cities were used to validate the new data structure. Although the new data structure is asymptotically not better than the conventional ones, it is empirically superior to the conventional ones in the size range covered in this study. Furthermore, the splay tree data structure, which is asymptotically the best tour representation, seems unlikely to beat the three-level tree data structure even when the number of cities is 10^7. It is also worthy of noting that some other local search heuristics such as 2-Opt and 3-Opt can get benefits from using the new data structure.

Acknowledgements

The authors wish to thank D. Applegate, R. Bixby, V. Chvatal, and W. Cook for making their code available; D.S. Johnson and L.A. McGeoch for organizing the 8th DIMACS Challenge on the TSP; G. Reinelt for providing the benchmarks in the TSPLIB.

References

1) D. Applegate, R. Bixby, V. Chvatal, and W. Cook, "TSP Cuts Which Do Not Conform to the Template Paradigm", *Computational Combinatorial Optimization*, Springer, pp. 261-304, 2001.
2) S. Lin and B. Kernighan, "Effective Heuristic Algorithm for the Traveling Salesman Problem," *Operation Research*, vol. 21, pp. 498-516, 1973.
3) D. Applegate, W. Cook, and A. Rohe, "Chained Lin-Kernighan for Large Traveling Salesman Problems," *INFORMS Journal of Computing*, vol. 15, no. 1, pp. 82-92, 2003.
4) O. Martin, S. W. Otto, and E. W. Felten, "Large-Step Markov Chains for the Traveling Salesman Problem," *Complex Systems*, vol. 5, no. 3, pp. 299-326, 1991.
5) D. S. Johnson and L. A. McGeoch, "The Traveling Salesman Problem: A Case Study in Local Optimization," *Local Search in Combinatorial Optimization*, John Wiley & Sons, pp. 215-310, 1997.
6) S. Jung and B. Moon, "The Natural Crossover for the 2D Euclidean TSP," *Proc. Genetic and Evolutionary Computation Conf. 2001*, pp. 1003-1010, 2001.
7) P. Merz and B. Freisleben, "Memetic Algorithms for the Traveling Salesman Problem," *Complex Systems*, vol. 13, no. 4, pp. 297-345, 2001.
8) R. Baraglia, J. I. Hidalgo, and R. Perego, "A Hybrid Heuristic for the Traveling Salesman Problem," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 6, pp. 613-622, 2001.
9) H. D. Nguyen, I. Yoshihara, K. Yamamori, and M. Yasunaga, "Greedy Genetic Algorithms for Symmetric and Asymmetric TSPs," *IPSJ Transactions on Mathematical Modeling and Its Applications*, vol. 43, no. SIG 10 (TOM 7), pp. 165-175, 2002.
10) M. L. Fredman, D. S. Johnson, L. A. McGeoch, and G. Ostheimer, "Data Structures for Traveling Salesmen," *Journal of Algorithms*, vol. 18, pp. 432-479, 1995.
11) M. Chrobak, T. Szymacha, and A. Krawczyk, "A Data Structure Useful for Finding Hamiltonian Cycles," *Theoretical Computer Science*, vol. 71, pp. 419-424, 1990.
12) G. Reinelt, "TSPLIB - A Traveling Salesman Problem Library," *ORSA J. on Computing*, vol. 3, no. 4, pp. 376-384, 1991. Available at http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB
13) D. S. Johnson and L. A. McGeoch, "Experimental Analysis of Heuristics for the STSP," *The Traveling Salesman Problem and Its Variations*, Kluwer Academic Publishers, pp. 369-443, 2002.
14) K. T. Mak and A. J. Morton, "A modified Lin-Kernighan Traveling Salesman Heuristic," *Operations Research Letters*, vol. 13, pp. 127-132, 1993.
15) J. L. Bentley, "Experiments on Traveling Salesman Heuristics," *Proc. 1st Annual ACM-SIAM Sym. On Discrete Algorithms*, pp. 91-99, 1990.
16) M. Held and R. M. Karp, "The Traveling Salesman Problem and Minimal Spanning Trees," *Operation Research*, vol. 18, pp. 1138-1162, 1970.
17) M. Held and R. M. Karp, "The Traveling Salesman Problem and Minimal Spanning Trees: Part II," *Mathematical Programming*, vol. 1, pp. 6-25, 1971.