A Local Search Algorithm Efficient for Sparse Instances of the Linear Ordering Problem

Celso Satoshi Sakuraba and Mutsunori Yagiura

Department of Computer Science and Mathematical Informatics
Graduate School of Information Science, Nagoya University
Furocho, Chikusaku, Nagoya 464-8603, Japan
sakuraba@al.cm.is.nagoya-u.ac.jp
yagiura@nagoya-u.jp

Abstract: Given a directed graph with n vertices, m edges and costs on the edges, the linear ordering problem consists of finding a permutation of the vertices such that the total cost of the reverse edges is minimized. We present a local search algorithm named LIST for the neighborhood of the *insert* operation, which can search the whole neighborhood in $O(n+m)$ time. Computational experiments show good results for sparse instances when compared with another method proposed in the literature.

Keywords: linear ordering problem, local search

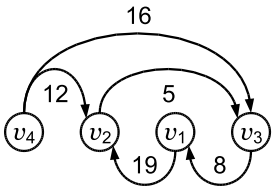
1 Introduction

Given a directed graph $G = (V, E)$ with a vertex set V ($|V| = n$), an edge set $E \subseteq V \times V$ and a cost c_{uv} for each edge (u, v), the linear ordering problem (LOP) consists of finding a permutation of vertices that minimizes the total cost of the reverse edges. We assume without loss of generality that $c_{uv} > 0$ holds for all $(u, v) \in E$ and that if we regard G as an undirected graph, it is connected (which implies $m \geq n - 1$). For convenience, we also assume $c_{uv} = 0$ for all $(u, v) \notin E$. Denoting the permutation by $\pi : \{1, \ldots, n\} \to V$, where $\pi(i) = v$ (equivalently, $\pi^{-1}(v) = i$) signifies that v is the ith element of π, the total cost of the reverse edges is formally defined as follows:

$$Cost(\pi) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c_{\pi(j)\pi(i)}. \tag{1}$$

Another representation of the LOP consists of finding a permutation of n indices such that when the rows and columns of an $n \times n$ matrix are permuted with it (n.b. the same permutation is used for rows and columns), the sum of the values in the upper triangle is maximized. The equivalence of the two representations is immediate, e.g., by regarding the matrix as the adjacency matrix of G, as can be seen in the example of Figure 1.

The LOP has a number of real world applications in various fields [5], among which the most widely known is the triangularization of input-output matrices, which allows economists to analyze the economical stability of a certain region. Known as an NP-hard problem [2], the LOP has been vastly studied in the literature since the appearance of the first paper about it [3],



	4	2	1	3
4	0	12	0	16
2	0	0	0	5
1	0	19	0	0
3	0	0	8	0

Figure 1: Graph and matrix representations of a solution of the LOP with cost 19+8=27

and many exact and heuristic methods have been proposed to solve it. Good literature reviews about the LOP are given in [4] and [8].

Among the heuristic approaches, there are a number of proposed metaheuristics to handle the LOP, such as tabu search [7], scatter search [1], variable neighborhood search [4] and genetic algorithm [6]. Such metaheuristics make use of local search methods to refine the quality of their solutions. Local search is a procedure that starts from an initial solution π_0 and repeats replacing it with a better solution in its neighborhood until no better solution is found. The neighborhood of a solution π is the set of solutions that can be obtained by applying an operation over π. A solution with no better solution in its neighborhood is called locally optimal.

The most widely known neighborhoods for the LOP are the ones given by the following operations:

- *insert*: taking one vertex from a position i and inserting it after (before) the vertex in position j for $i < j$ $(i > j)$;
- *interchange*: exchanging the vertices in positions i and j.

Huang and Lim [6] show that although any solution that can be improved by *interchange* can be improved by *insert*, not every solution that can be improved by *insert* can be improved by *interchange*. As expected, Schiavinotto and Stützle [8] affirm that *interchange* has experimentally worse results than *insert*, and all the metaheuristic algorithms cited before make use of the *insert* operation. The algorithms proposed in this work make use of the *insert* operation as well.

Let π' be the permutation obtained from a permutation π by inserting the vertex at position i into position j. The difference in cost of this is given by:

$$Cost(\pi') - Cost(\pi) = \begin{cases} \sum_{k=i+1}^{j} (c_{\pi(i)\pi(k)} - c_{\pi(k)\pi(i)}), & i < j \\ \sum_{k=j}^{i-1} (c_{\pi(k)\pi(i)} - c_{\pi(i)\pi(k)}), & i > j. \end{cases} \tag{2}$$

This cost difference can be calculated in $O(n)$ time, which makes a straightforward search through the insert neighborhood possible in $O(n^3)$ time, where a search through the neighborhood denotes the task of finding an improved solution or concluding that no such solution exists (i.e., the current solution is locally optimal). However, if we make the search in an ordered way, taking one vertex at a time and sequentially calculating the cost of inserting it in consecutive positions, the calculation can be made in constant order of time and the search through the neighborhood can be done in $O(n^2)$ time. This result, presented by Schiavinotto and Stützle [8], is the best one found so far concerning local search methods for the LOP.

Given the relevance of the insert neighborhood for the LOP, this paper presents an algorithm for the search through it. This algorithm, named LIST, can make the search through the whole neighborhood in $O(m)$ time, where $m = |E|$. Computational results obtained by experiments are presented and compared with other results in the literature.

The following section introduces the LIST algorithm, and Section 3 presents experimental results obtained with its application. The last section presents the conclusions of our work.

2 The LIST Algorithm

The LIST algorithm proposed here uses a doubly linked list data structure corresponding to a solution of the problem. One list is built for each vertex (represented in the head of the list), and each cell of the list except for the head represents a vertex adjacent to it. Cells in a list are ordered according to the order of the current solution. The information kept in each cell (except for the head) is the index of the vertex and the cost of the edge connecting it with the vertex represented in the head of the list. These costs are represented as negative if the edge is a reverse one and as positive otherwise, as shown in Figure 2. The total memory space necessary to keep this data structure is $O(m)$.

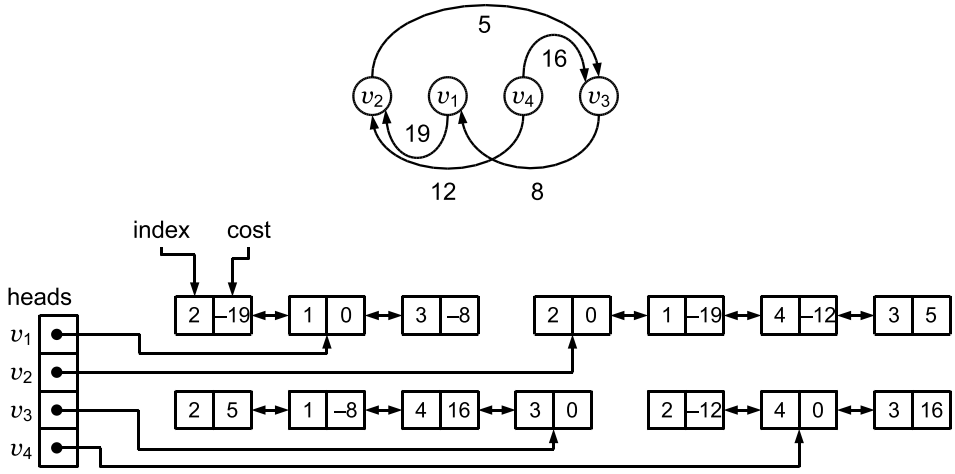


Figure 2: The linked lists of the LIST algorithm for solution $\pi = (v_2, v_1, v_4, v_3)$

2.1 Construction of the lists

To build the whole LIST structure from a given list of edges (u, v) and an initial permutation π_0, we first sort the edges in the nondecreasing order of $\pi_0^{-1}(u)$, breaking ties with the nondecreasing order of $\pi_0^{-1}(v)$. That is, the sorting is made such that groups containing edges incident from the same vertex are ordered according to π_0, and edges inside each group are ordered by the index of the vertex they are incident to according to π_0 as well (see (2) in Figure 3). This sorting can be done in $O(m)$ time using radix sort. Note that the given list of edges may include parallel edges (edges that are incident from the same vertex and incident to the same vertex).

In the second step, we create for each vertex u a list of vertices v satisfying $(u, v) \in E$. Each cell of this list contains the index of the vertex v and the value of c_{uv}, where c_{uv} is the

sum of the costs of parallel edges from u to v if there are such edges. To create such lists, we start from empty lists and insert cells one by one to the tail of each list according to the order they are sorted, where the information of parallel edges (that appear consecutively in the sorted list of edges) are merged in this stage by summing up their costs instead of creating duplicate cells with the same v. Then the resulting linked list for each head u maintains the order of the permutation π_0 (see (3) in Figure 3).

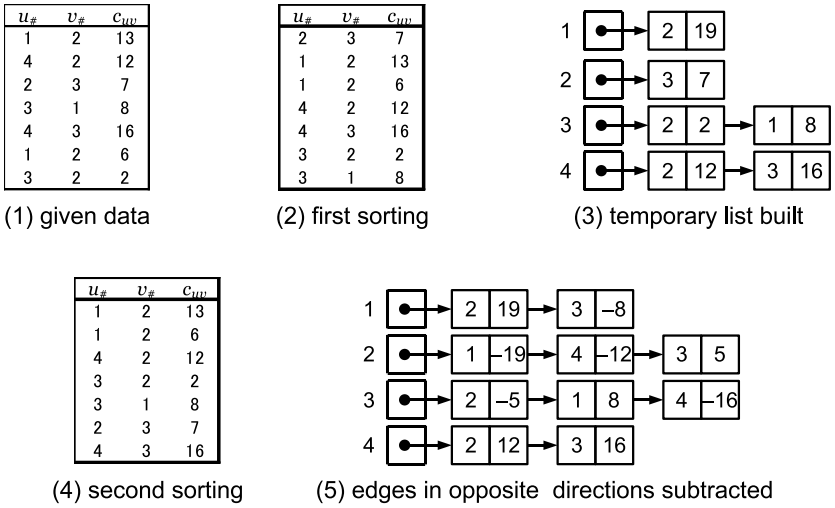


Figure 3: Procedure to build the data structure of Figure 2 ($\pi_0 = (v_2, v_1, v_4, v_3)$)

Then, we sort the edges (u, v) in nondecreasing order of $\pi_0^{-1}(v)$, breaking ties with the nondecreasing order of $\pi_0^{-1}(u)$ (see (4) in Figure 3). For each edge (u, v), following the order the edges are sorted, we subtract cost c_{uv} from the cell corresponding to (v, u) in the list with head v if such a cell exists; otherwise, we create a new cell for (v, u) with cost $-c_{uv}$ and insert it at the appropriate position according to π_0 in the list of v (see (4) in Figure 3). By keeping a pointer to the position of the cell u most recently created or modified, this step can be done in $O(m)$ time.

Finally, we add a head to the list corresponding to each vertex at the appropriate position according to π_0. The head is inserted by visiting the list from its leftmost cell until we reach the desired position, with each cell visited in the way having its cost multiplied by -1. The initial LIST data structure built by the steps in Figure 3 is the one shown in Figure 2.

Consequently, the LIST data structure can be constructed from scratch in $O(m)$ time for a given permutation π_0 and a list of edges with their costs.

2.2 Neighborhood search

From the data structure of LIST corresponding to a solution π, we can easily calculate the difference in the total cost generated by inserting a vertex u in the position of an adjacent vertex v. Supposing that u is before v in π, we start from the head of the list corresponding to u and follow the list to the right until we find the cell corresponding to v, keeping the sum of the costs of the cells visited, including the cost of v itself. This sum is the difference in cost caused by inserting u in the position of v.

Using this calculation, the difference in cost generated by inserting a vertex u in the position of a vertex corresponding to the cell right next to the head of the list of u can be calculated in constant order of time just by taking the cost of that cell. By repeating this process following the sequence of the list of u and adding the cost of the next cell to the difference in cost calculated for the current cell, we can calculate the costs of all solutions obtainable by inserting u into those positions of adjacent vertices in $O(d_u)$ time, where d_u is the degree of u (i.e., the number of edges incident to and from u). Note that it is not necessary to compute the costs of those solutions generated by inserting u into other positions, since the cost of such a solution must be the same as one of the solutions whose cost is computed in the above procedure. Hence, we can find a better solution (or can conclude that there is no such a solution) in $O(\sum_{u \in V} d_u) = O(m)$ time.

If a permutation with smaller cost than π is found by inserting a vertex u into a new position, it is necessary to update the related parts of the data structure. This update is made by the following operations:

- For the list having u as its head, change its head position to the new one and change the sign of the costs in the cells between the initial and the new positions of the head.
- For the list of each vertex adjacent to u, find the cell corresponding to u and update its position. If the new position is after (before) and the initial position is before (after) the list head, multiply the cost of the moved cell by -1.

The first operation takes $O(d_u)$ time, and the second one takes $O(d_v)$ time for each v adjacent to u. As the sum of d_u and the values of d_v for those adjacent vertices v is not more than $\sum_{v \in V} d_v = 2m$, the update of the data structure can be done in $O(m)$ time.

The updated graph of Figure 2 after the insertion of v_4 into position 1 and the corresponding linked lists are presented in Figure 4. Observe that there is no change in the list of v_1, which is not adjacent to v_4

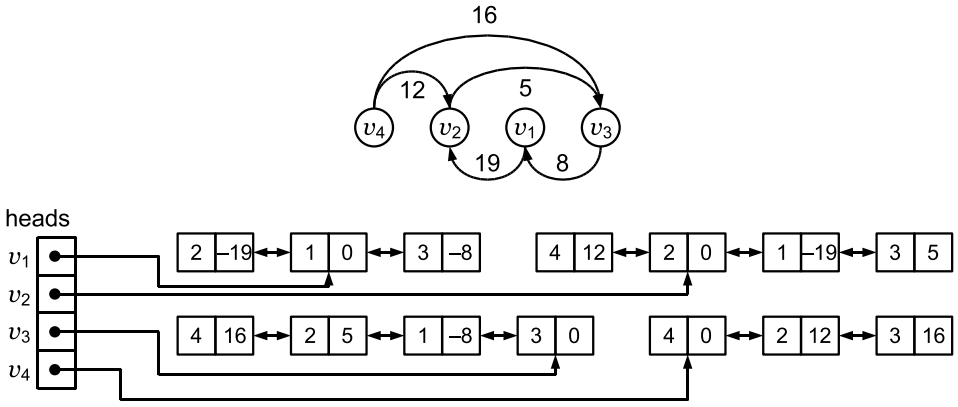


Figure 4: Solution in Figure 2 after the insertion of v_4 into position 1

To evaluate the computation necessary to search the neighborhood, we use the following definition proposed by Yagiura and Ibaraki [9]:

Definition. *The necessary computation to determine a move in a neighborhood is called* one-round*. This computation includes finding an improved solution in the neighborhood and updating*

all the necessary data structures, or concluding that there is no better solution in the neighborhood.

Using this definition and the analysis presented above, we can state the following:

Theorem. *The worst case one-round time for the LIST algorithm is $O(m)$. The data structure for a given permutation can be built in $O(m)$ time as well.*

3 Computational Results

The performance of the LIST algorithm was evaluated using a set of randomly generated instances. Instances of sizes (number of vertices) between 500 and 8000 were prepared, and five values of density (probability that an edge between any two vertices exists) were considered. For each instance class (combination of size and density), five instances were generated by randomly choosing edge costs from the integers in the interval $[1, 99]$ using Mersenne Twister presented in http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html.

To a better evaluation of LIST, we compare its one-round time with that of the method described in Schiavinotto and Stützle [8], which will be referred here as SCST. Codes were written in C language and tests were conducted on a Intel Xeon with a 3.0 GHz processor and 8GB RAM.

Table 1 presents the results. Values in the table denote the average one-round time in seconds to search the whole neighborhood of the insert operation.

Table 1: LIST Algorithm Results

density	1%		5%		10%		50%		100%	
n	SCST	LIST	SCST	LIST	SCST	LIST	SCST	LIST	SCST	LIST
500	0.0031	0.00006	0.0029	0.0009	0.0029	0.0017	0.0027	0.0135	0.0028	0.0404
1000	0.0363	0.00023	0.0355	0.0012	0.0353	0.0082	0.0348	0.0809	0.0347	0.1900
2000	0.1891	0.00104	0.1859	0.0243	0.1846	0.0561	0.1821	0.3606	0.1806	0.8599
3000	0.4466	0.00751	0.4384	0.0636	0.4357	0.1351	0.4288	0.8594	0.4271	2.0401
4000	0.8371	0.01975	0.8306	0.1207	0.8238	0.2557	0.8998	1.5324	0.8872	4.6174
8000	4.7873	0.09619	4.7200	0.5111	4.7350	1.0851	4.7516	8.6897	4.8053	23.0210

As expected, although the results of SCST does not change much against the density, the values for LIST changes significantly. LIST is faster than SCST for sparser instances, with density between 1 and 10%. However, for the instances with density 50%, LIST takes around twice the time of SCST.

4 Conclusions

In this paper, we studied local search algorithms for the linear ordering problem, developing an algorithm to make the search in the neighborhood of insert operation faster for sparse instances.

The data structure of the LIST algorithm proposed uses $O(m)$ memory space, and the whole data structure for it can be built in $O(m)$ time. One-round time of the LIST algorithm is $O(m)$, and computational experiments with random instances showed good results for sparse instances with density up to 10% when compared to the results of an algorithm presented in the literature.

LIST does not present much difficulties in its implementation, and can be easily used as a part of metaheuristic algorithms, being strongly recommended for sparse instances.

As for future works, we intend to develop algorithms that can obtain good results even for dense instances and use these algorithms as parts of metaheuristics and other methods to obtain better results for the LOP.

Acknowledgments

The authors are grateful to Professor Takao Ono for his valuable comments in implementation issues. This research is partially supported by a Scientific Grant-in-Aid from the Ministry of Education, Culture, Sports, Science and Technology of Japan and by The Hori Information Science Promotion Foundation.

References

[1] V. Campos, F. Glover, M. Laguna, R. Martí, An experimental evaluation of a scatter search for the linear ordering problem, *Journal of Global Optimization* **21** (2001) 397–414.

[2] S. Chanas, P. Kobylasński, A new heuristic algorithm solving the linear ordering problem, *Computational Optimization and Applications* **6** (1996) 191–205.

[3] H.B. Chenery, T. Watanabe, International comparisons of the structure of production, *Econometrica* **26** (1958) 487–521.

[4] C.G. Garcia, D. Pérez-Brito, V. Campos, R. Martí, Variable neighborhood search for the linear ordering problem, *Computers & Operations Research* **33** (2006) 3549–3565.

[5] M. Grötschel, M. Jünger, G. Reinelt, A cutting plane algorithm for the linear ordering problem, *Operations Research* **32** (1984) 1195–1220.

[6] G.F. Huang, A. Lim, Designing a hybrid genetic algorithm for the linear ordering problem, *Genetic and Evolutionary Computation - GECCO 2003, proceedings* (2003) 1053–1064.

[7] M. Laguna, R. Martí, V. Campos, Intensification and diversification with elite tabu search solutions for the linear ordering problem, *Computers & Operations Research* **26** (1999) 1217–1230.

[8] T. Schiavinotto, T. Stützle, The linear ordering problem: Instances, search space analysis and algorithms, *Journal of Mathematical Modelling and Algorithms* **3** (2004) 367–402.

[9] M. Yagiura, T. Ibaraki, Analyses on the 2 and 3-flip neighborhoods for the MAX SAT, *Journal of Combinatorial Optimization* **3** (1999) 95–114