INTERNATIONAL TRANSACTIONS IN OPERATIONAL RESEARCH

I F O R S International Federation of Operational Research Societies

INTERNATIONAL TRANSACTIONS IN OPERATIONAL RESEARCH

Intl. Trans. in Op. Res. 17 (2010) 711–737
DOI: 10.1111/j.1475-3995.2010.00778.x

Efficient local search algorithms for the linear ordering problem

Celso S. Sakuraba and Mutsunori Yagiura

Department of Computer Science and Mathematical Informatics, Graduate School of Information Science, Nagoya University, Furocho, Chikusaku, Nagoya 464-8603, Japan
E-mail: sakuraba@al.cm.is.nagoya-u.ac.jp [Sakuraba]; yagiura@nagoya-u.jp [Yagiura]

Received 20 January 2009; received in revised form 17 March 2010; accepted 24 March 2010

Abstract

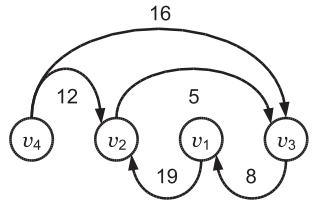
Given a directed graph with n vertices, m edges and costs on the edges, the linear ordering problem consists of finding a permutation π of the vertices so that the total cost of the reverse edges is minimized. We present two local search algorithms, named LIST and TREE, for the neighborhood of the *insert* move, which can handle larger instances than existing methods. LIST is simpler and can search the whole neighborhood in $O(m)$ time and TREE performs the neighborhood search in $O(n+\Delta\log\Delta)$ time, where Δ represents the maximum vertex degree. Computational experiments show good results for sparse instances using LIST, while TREE presents the best results independent of the density of the instance.

Keywords: linear ordering problem; local search; balanced search tree.

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, i.e., edges directed from a vertex u to a vertex v with v being a vertex in a position before u in the permutation. 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 $m\geq n-1$, where $m=|E|$). For convenience, we also assume $c_{uv}=0$ for all $(u,v)\notin E$. Denoting the permutation by $\pi:\{1,\ldots,n\}\rightarrow V$, 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:

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



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

Fig. 1. Graph and matrix representations of a solution of the linear ordering problem (LOP) with cost 19+8 = 27.

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 (i.e., values above the diagonal) 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 Fig. 1. In the figure, each vertex v_i corresponds to column and row i in the matrix. Based on studies presented in Grötschel et al. (1984, 1985), the LOP (in its matrix version) can be formulated as the following integer program:

$$\begin{aligned}
\text{maximize} \quad & \sum_{i \in V} \sum_{\substack{j \in V \\ j \neq i}} c_{ij} x_{ij} \\
\text{subject to} \quad & x_{ij} + x_{ji} = 1, && \forall i, j \in V, i \neq j, \\
& x_{ij} + x_{jk} + x_{ki} \leq 2, && \forall i, j, k \in V, i \neq j \neq k \neq i, \\
& x_{ij} \in \{0, 1\}, && \forall i, j \in V.
\end{aligned}$$

Throughout the remainder of this paper we deal with the graph formulation of the LOP (the one that minimizes the cost of the reverse edges) unless otherwise stated.

The LOP has a number of real world applications in various fields (Grötschel et al., 1984), among which the most widely known application 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 (Karp, 1972; Garey and Johnson, 1979),[1] the LOP has been extensively studied in the literature since the appearance of the first paper about it (Chenery and Watanabe, 1958), and many exact and heuristic methods have been proposed to solve it. Good literature reviews about the LOP are given by Schiavinotto and Stützle (2004) and Charon and Hudry (2007).

Among the heuristic approaches, there are a number of metaheuristics to handle the LOP, such as tabu search (Laguna et al., 1999), scatter search (Campos et al., 2001), iterated local search (Schiavinotto and Stützle, 2003), genetic algorithm (Huang and Lim, 2003), memetic algorithm (Schiavinotto and Stützle, 2004) and variable neighborhood search (Garcia et al., 2006). 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_{init} and repeatedly replaces 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

[1]The proof of NP-completeness was given for the feedback arc set problem, which is equivalent to the LOP.

with no better solution in its neighborhood is called locally optimal. Algorithm 1 shows the framework of a local search.

Algorithm 1. Framework of the local search
input: an initial solution π_{init}
output: a locally optimal solution π and its cost $cost(\pi)$
$\pi \leftarrow \pi_{\text{init}}$, $cost \leftarrow \infty$
while $cost(\pi) < cost$ **do**
  $cost \leftarrow cost(\pi)$
  search the neighborhood of π for a solution π with $cost(\pi') < cost$
  if such a solution π' is found **then**
    $\pi \leftarrow \pi'$
  end if
end while
output π and $cost(\pi)$

There are two move strategies that are often used with local search. The first is the best move strategy, which consists of searching through the whole neighborhood and moving to the neighbor that has the minimum cost. The second is the first admissible move strategy, by which the algorithm moves to a new solution as soon as a solution with a smaller cost is found.

We define *search through the neighborhood* as the task of finding an improved solution or concluding that no such solution exists (i.e., the current solution is locally optimal). The computation time necessary to perform such a task, including the time to update relevant data structures, is called *one-round time* (Yagiura and Ibaraki, 1999). This time corresponds to one round of the while loop in Algorithm 1.

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 (respectively, before) the vertex in position j for $i<j$ (respectively, $i>j$);
- *interchange*: exchanging the vertices in positions i and j.

Huang and Lim (2003) 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 (2004) 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 in position i into position j. The difference in cost of this operation is given by

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

This cost difference generated by an *insert* can be calculated in $O(n)$ time, which makes a straightforward search through the insert neighborhood possible in $O(n^3)$ time.

If we conduct the search in an ordered way taking one vertex at a time and sequentially calculating the cost of inserting it into consecutive positions, the calculation for each insert position can be done in constant order of time, reducing the one-round time to $O(n^2)$. This algorithm, presented by Schiavinotto and Stützle (2004) and referred to as SCST in this paper, is the best algorithm found so far vis-à-vis the one-round time of local search methods for the LOP.

Given the relevance of the *insert* neighborhood for the LOP, in this paper we present two algorithms for the search through it. Both algorithms were developed to perform a local search like the algorithm proposed in Schiavinotto and Stützle (2004) with a smaller one-round time. The first algorithm, named LIST, can search the neighborhood in $O(m)$ time. The second algorithm, named TREE, can search the whole neighborhood in $O(n+\Delta\log\Delta)$ time, where Δ is the maximum degree of the graph (denoting by d_v, the degree of a vertex v, i.e., the number of vertices incident to and from v, $\Delta = \max_{v\in V} d_v$). Computational experiments with both algorithms are presented and the results are compared with those of the $O(n^2)$ time algorithm cited previously. The comparison shows that TREE is the most efficient algorithm among the three, being more than a hundred times faster than the other methods for large instances, and that LIST has good results for sparse instances.

The following two sections introduce the LIST and TREE algorithms, and Section 4 presents experimental results obtained by their implementation. The last section presents the conclusions of our work.

2. The LIST algorithm

The LIST algorithm proposed in this section uses a doubly linked list data structure corresponding to a solution of the problem. Let $N(v)$ be the set of all vertices adjacent to a vertex v (i.e., $N(v) = \{u \in V | (u, v) \in E \text{ or } (v, u) \in E\}$). One list is built for each vertex v (represented in the head of the list) and each cell of the list except for the head represents a vertex $u \in N(v)$. 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 u and the cost of the edge connecting it with v. These costs are represented as negative if the edge is a reverse one and as positive otherwise, as shown in Fig. 2. The total memory space necessary to keep this data structure is $O(m)$.

The next two subsections describe how to build the lists from scratch and how to use them to perform a local search.

2.1. Construction of the lists

To build the whole LIST data structure from a given list of edges (u, v) and an initial permutation π_{init}, we first sort all the edges in the nondecreasing order of $\pi_{\text{init}}^{-1}(u)$, breaking ties with the nondecreasing order of $\pi_{\text{init}}^{-1}(v)$. In other words, the sorting is made such that groups containing edges incident from the same vertex are ordered according to π_{init}, and edges inside each group are

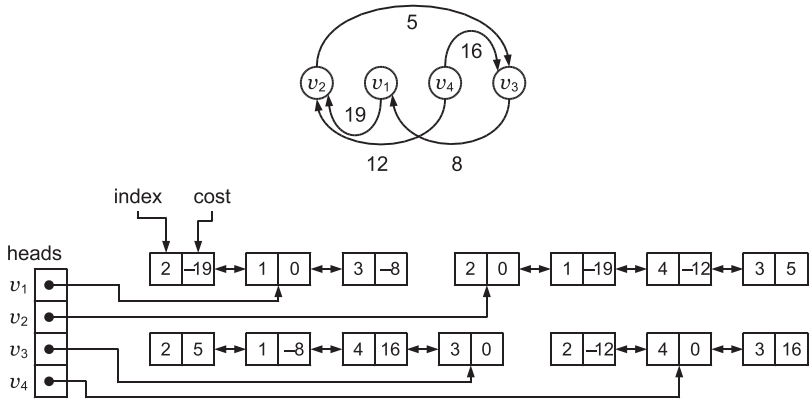


Fig. 2. The linked lists of the LIST algorithm for a solution $\pi = (v_2, v_1, v_4, v_3)$.

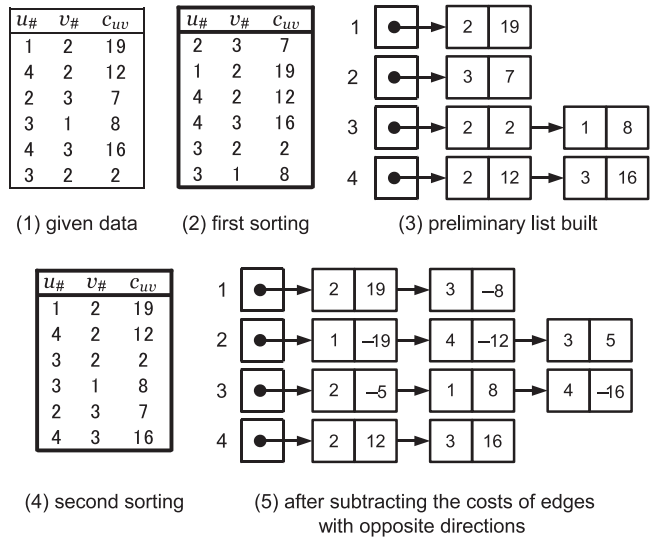


Fig. 3. Procedure to build the data structure of Fig. 2 ($\pi_{\text{init}} = (v_2, v_1, v_4, v_3)$).

ordered by the index of the vertex to which they are incident according to π_{init} as well (see (2) in Fig. 3). This sorting can be done in $O(m)$ time using radix sort.

In the second step, we create for each vertex u a list of vertices $v \in N(u)$. Each cell of this list contains the index of the vertex v and the value of c_{uv}. 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. Then, the resulting linked list for each head u maintains the order of the permutation π_{init} (see (3) in Fig. 3).

Then, we sort the edges (u, v) in nondecreasing order of $\pi_{\text{init}}^{-1}(v)$, breaking ties with the nondecreasing order of $\pi_{\text{init}}^{-1}(u)$ (see (4) in Fig. 3). For each edge (u, v), following the order the edges are sorted, we subtract cost c_{uv} from the cell corresponding to u in the list with head v if such

a cell exists; otherwise, we create a new cell for u with cost $-c_{uv}$ and insert it at the appropriate position according to π_{init} in the list of v (see (5) in Fig. 3). By this procedure, if we have edges (u, v) and (v, u) for a pair of vertices u and v with costs satisfying $c_{uv} \geq c_{vu}$ without loss of generality, then edge (v, u) is deleted, and the cost of (u, v) is modified to $c_{uv} - c_{vu}$. In this case, we keep the value of c_{vu} to be added to the cost of the final solution. In the final list (5) of the example of Fig. 3, we have cells representing edge (2, 3) with cost $c_{23} - c_{32} = 5$ in the lists of 2 and 3, and we keep the value of $c_{32} = 2$ to be added to the cost of the final solution. By keeping a pointer to the position of the cell for 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_{init}. 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 Fig. 3 is the one shown in Fig. 2.

Consequently, the LIST data structure can be constructed from scratch in $O(m)$ time for a given permutation π_{init} 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 v into the position of a vertex $u \in N(v)$. Supposing $\pi^{-1}(v) < \pi^{-1}(u)$ (the other case is symmetric), we start from the head of the list corresponding to v and follow the list to the right until we find the cell corresponding to u, keeping the sum of the costs of the cells visited, including the cost of u itself. This sum is the difference in cost caused by inserting v into the position of u.

Using this calculation, the difference in cost incurred by inserting a vertex v into the position of a vertex corresponding to the cell right next to the head of the list of v 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 v 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 v into the positions of adjacent vertices in $O(d_v)$ time, where d_v is the degree of v (i.e., the number of edges incident to and from v). An example of this calculation is shown in Fig. 4. Note that it is not necessary to compute the costs of the solutions generated by inserting v into other positions, because 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 solution) in $O\left(\sum_{v \in V} d_v\right) = O(m)$ time.

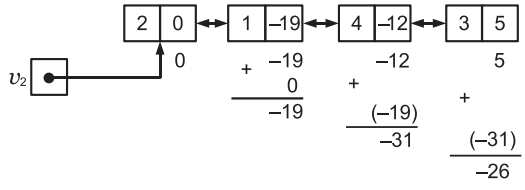


Fig. 4. Calculation of the difference in cost of inserting v_2 into consecutive positions of the solution in Fig. 2; inserting v_2 into the position of v_4 decreases the cost by 31.

Once we find a solution with smaller cost than π, it is necessary to update the data structure to make it correspond to this new solution in order to continue the search from it. This update is done through the following operations:

- For the list having v 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 $u \in N(v)$, find the cell corresponding to v and update its position. If the new position is after (respectively, before) and the initial position is before (respectively, after) the list head, multiply the cost of the moved cell by -1.

The first operation takes $O(d_v)$ time and the second one takes $O(d_u)$ time for each $u \in N(v)$. As the sum of d_v and the values of d_u for $u \in N(v)$ is not bigger 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 Fig. 2 after the insertion of v_4 into the position of v_2 and the corresponding linked lists are presented in Fig. 5. Observe that there is no change in the list of v_1, which is not adjacent to v_4.

Based on the analysis presented above, we can state the following:

Theorem 1. *The one-round time for the LIST algorithm is $O(m)$. The data structure for a given permutation can also be built from scratch in $O(m)$ time.*

Note that because $m = O(n^2)$, the one-round time of the LIST algorithm is never asymptotically slower than that of SCST. Algorithm 2 shows the framework of the LIST algorithm. The time necessary to perform each operation, when different from constant, is represented in parentheses.

Algorithm 2. LIST algorithm
input: graph G and an initial solution π_{init}
output: a locally optimal solution π and its cost $cost(\pi)$
$\pi \leftarrow \pi_{\text{init}}$, $cost \leftarrow \infty$
create the lists for all vertices as described in Section 2.1 $(O(m))$
while $cost(\pi) < cost$ **do**
  $cost \leftarrow cost(\pi)$

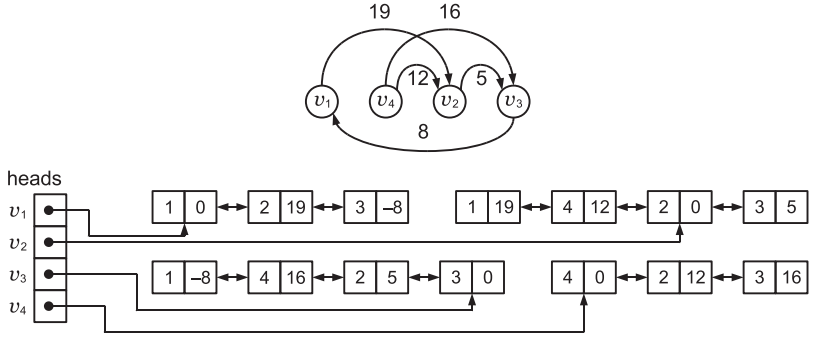


Fig. 5. Solution in Fig. 2 after the insertion of v_4 into position 1.

search the lists to find a vertex v that can be inserted in a different position generating a solution π' with $cost(\pi') < cost$ ($O(m)$)
if such a vertex v and a solution π' is found **then**
$\pi \leftarrow \pi'$ ($O(n)$)
update the lists of vertex v and of all vertices $u \in N(v)$ ($O(m)$)
end if
end while
output π and $cost(\pi)$

3. The TREE algorithm

The TREE algorithm presented in this section uses a balanced search tree data structure to make the search through the insert neighborhood efficient. Our implementation is based on a 2–3 tree, i.e., a tree such that all the inner nodes have 2 or 3 children and all the leaves have the same depth (Aho et al., 1974). A tree is built for each vertex, and we use the tree for a vertex v to calculate the cost of solutions obtained by inserting v into different positions of π. See Fig. 6 for an example of the whole data structure of TREE.

Let $\pi_v : \{1, 2, \ldots, d_v\} \to N(v)$ be the permutation of the vertices $u \in N(v)$ having the same order as π, i.e., $\pi_v^{-1}(u) < \pi_v^{-1}(w) \Leftrightarrow \pi^{-1}(u) < \pi^{-1}(w)$ for any two vertices u and w in $N(v)$. For convenience, dummy nodes v_0 and v_{n+1} are added to the beginning and to the end of π and of each π_v ($\pi(0) = v_0$ and $\pi(n+1) = v_{n+1}$; $\pi_v(0) = v_0$ and $\pi_v(d_v + 1) = v_{n+1}$ for all $v \in V$).

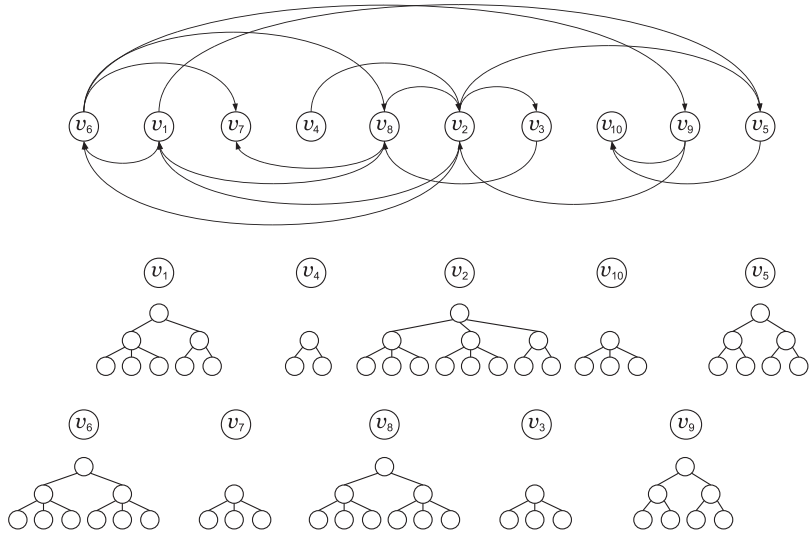


Fig. 6. An example of the whole TREE data structure corresponding to the permutation $\pi = (v_6, v_1, v_7, v_4, v_8, v_2, v_3, v_{10}, v_9, v_5)$; the costs of the edges in the graph are omitted.

In our data structure, each leaf l in the tree for a vertex v corresponds to a gap between two consecutive vertices of π_v. An example for a vertex v_2 with $\pi = (v_6, v_1, v_7, v_4, v_8, v_2, v_3, v_{10}, v_9, v_5)$ and $N(v_2) = \{v_6, v_1, v_4, v_8, v_3, v_9, v_5\}$ is shown in Fig. 7. The lower part of this figure shows the vertices in $N(v_2)$, with the costs of the edges connecting them with v_2, and the upper part shows the tree corresponding to v_2. It should be observed that vertices not adjacent to v_2 do not appear in the tree, because their relative position to v_2 has no influence on the cost of the solution. The values in the nodes of the tree are explained later.

To explain how our data structure works, we first define the cost of a vertex v in the current solution π. This cost corresponds to the sum of the costs of all reverse edges connected with v in the current solution and is given by

$$cost(v, \pi) = \sum_{i < \pi^{-1}(v)} c_{v,\pi(i)} + \sum_{i > \pi^{-1}(v)} c_{\pi(i),v}. \tag{3}$$

In Fig. 7, $cost(v_2, \pi) = 8 + 35 + 42 = 85$. The total cost of a solution π is given by half of the sum of $cost(v, \pi)$ for all $v \in V$. For the current solution, we keep a list of $cost(v, \pi)$ for all the vertices $v \in V$.

In the tree for a vertex v, each node x keeps a value $\gamma(x)$ (represented in the right bottom cell of each node of the tree in Fig. 7). For a pair of nodes (y, z), with y an ancestor of z, we define $P(y, z)$ as the set of all nodes contained in the unique path from node y to node z, including these two. We then define the value of a path between nodes y and z as $\gamma(P(y, z)) = \sum_{x \in P(y,z)} \gamma(x)$.

Let $c_v^{\mathrm{rev}}(l)$ be the cost incurred by inserting a vertex v into the position corresponding to a leaf l of the tree of v, i.e., the sum of the costs of reverse edges connected with v when the position of v is in the gap defined by l. We control $\gamma(x)$ so that $c_v^{\mathrm{rev}}(l) = \gamma(P(r_v, l))$ holds, where r_v is the root of the tree for v. The rules to achieve this are explained in the following subsections. Denoting by $\pi'(v, l)$ the solution obtained from π by inserting a vertex v into the position corresponding to a leaf l, we have $cost(\pi'(v, l)) - cost(\pi) = c_v^{\mathrm{rev}}(l) - cost(v, \pi)$.

Two other values are kept in each node x of the tree. The first value, $v_{\mathrm{name}(x)}$, carries the name of the vertex on the left of the gap represented by x if x is a leaf, or the value of $v_{\mathrm{name}}(y)$ of the rightmost $y \in C(x)$, where $C(x)$ is the set of children of x, if x is an inner node. In Fig. 7, $v_{\mathrm{name}}(x)$ is represented in the left cell of each node. By keeping the values of $v_{\mathrm{name}}(x)$ this way and using π^{-1}, we can find any leaf l in the tree of v by its $v_{\mathrm{name}}(l)$ in $O(\log d_v)$ time.

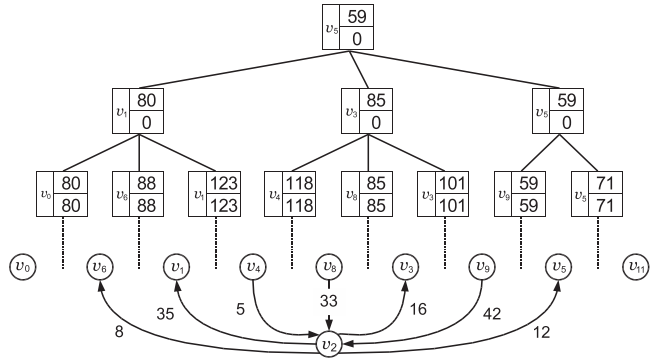


Fig. 7. The vertices adjacent to vertex v_2 and the tree corresponding to v_2.

The second value, $\gamma_{\min}(x)$, is equal to the minimum path value among the paths between x and one of the leaves in the subtree whose root is x, i.e., $\gamma_{\min}(x) = \min_{l \in L(x)} \gamma(P(x,l))$, where $L(x)$ is the set of leaves in the subtree of a node x. In Fig. 7, $\gamma_{\min}(x)$ is represented in the right upper cell of each node. For a leaf l, $\gamma_{\min}(l) = \gamma(l)$, and for the other nodes x, $\gamma_{\min}(x) = \gamma(x) + \min_{y \in C(x)} \gamma_{\min}(y)$.

From the definitions above, $\gamma_{\min}(r_v)$ is equal to the minimum value of $c_v^{\mathrm{rev}}(l)$ among all the leaves in the tree of v, and we can look for a solution with a smaller cost than the current one just by comparing the values of $\gamma_{\min}(r_v)$ and $cost(v,\pi)$ for all $v \in V$. If none of the trees of v have $\gamma_{\min}(r_v)$ smaller than $cost(v,\pi)$, we can conclude that the current solution π is locally optimal.

The number of leaves in the tree of each vertex v is equal to $d_v + 1$, and the number of inner nodes of a tree is less than the number of leaves. Moreover, each node of the trees carries a fixed amount of information. Hence, the total memory space necessary to keep this data structure is $O\left(\sum_{v \in V} (d_v + 1)\right) = O(m+n) = O(m)$.

3.1. Initialization

To build the trees from a list of edges (u, v) and an initial permutation π_{init}, we first make a list for each vertex v. Each cell in the list of v contains the information of a vertex $u \in N(v)$, and the cells are listed in the same order as π_{init}. In the cell of each u, we keep the index of the vertex u and the value $c_{vu} - c_{uv}$. The lists for all v can be built in $O(m)$ time by using a procedure similar to the one shown in the initialization of the LIST algorithm.

For the tree of each vertex v, we start with an empty tree and add leaves l one by one, with the first leaf having $v_{\mathrm{name}}(l) = v_0$ and $\gamma(l) = \gamma_{\min}(l) = \sum_{u \in V} (c_{uv} - c_{vu})$. Then, we scan the list of v, creating a leaf for each cell corresponding to a vertex u and inserting it to the right of the last inserted leaf l'. For each inserted leaf l corresponding to u, we set $v_{\mathrm{name}}(l) := u$ and $\gamma(l) := \gamma_{\min}(l) := \gamma(l') + c_{vu} - c_{uv}$. Inner nodes are created according to the insert operation for 2–3 trees. Because the values in the inner nodes can be calculated only by looking at the values in its children, each leaf can be inserted in the tree for v in $O(\log d_v)$ time. Hence, the total time to build the trees is $O(m \log \Delta)$.

This time complexity can be reduced by slightly modifying the above procedure as follows: in the tree for each vertex v, first create all the leaves, which takes $O(d_v)$ time, then create the inner nodes in the level above the leaves and then the ones in one level above, until the root is created. As each node can be created in constant order of time, the time to create the tree for each vertex v is $O(d_v)$ and the time to create all the trees is $O(m)$.

The list with the values of $cost(v,\pi_{\mathrm{init}})$ of all vertices can be build in $O(m)$ time by scanning the list of edges and comparing the positions of their end vertices using π_{init}^{-1}.

3.2. Search and update of the data structure

To conduct a search through the neighborhood of a solution π, we look at the $\gamma_{\min}(r_v)$ values in the roots of the trees for all vertices $v \in V$ and compare them with the values of $cost(v,\pi)$ that are kept in a list. This procedure can be done in $O(n)$ time.

Once we find a v that satisfies $\gamma_{\min}(r_v) < cost(v, \pi)$, which indicates that we can decrease the total cost by inserting v into a different position, we need to find the position into which v should be inserted. This position is in a gap corresponding to one of the leaves of the tree for v. To find this leaf l_{pos}, we look for the path with $\gamma(P(r_v, l_{\text{pos}})) = \gamma_{\min}(r_v)$ through the following procedure: start from $x := r_v$ and replace x with one of its children y satisfying $\gamma_{\min}(y) + \gamma(x) = \gamma_{\min}(x)$ (which means that the leaf we are looking for is in the subtree that has y as its root) until x is replaced with a leaf; then, let $l_{\text{pos}} := x$.

Then, we know that the cost of the current solution π can be reduced by $cost(v, \pi) - c_v^{\text{rev}}(l_{\text{pos}})$ if we insert v into one of the positions corresponding to the leaf l_{pos}. For simplicity, in our algorithm, we set this position to the one immediately after vertex u with $u = v_{\text{name}}(l_{\text{pos}})$.

In the example of Fig. 7, $cost(v_2, \pi) = 85$ and $\gamma_{\min}(r_{v_2}) = 59$, which means that if we remove v_2 from the current position between v_8 and v_3 and insert it into a different position, we can reduce the cost by 85–59 = 26. To find the position where v_2 should be inserted, we start from the root and follow the nodes x with $\gamma_{\min}(x) = 59$ until we get to the leaf with $v_{\text{name}}(x) = v_9$.

After inserting v immediately after u, we update the trees for all the vertices $w \in N(v)$. Suppose $\pi^{-1}(v) < \pi^{-1}(u)$ holds before the insertion (the other case is discussed later). In the tree for each w, we look for the leaves l_v with $v_{\text{name}}(l_v) = v$ and l_u with $v_{\text{name}}(l_u) = u$ or such that $v_{\text{name}}(l_u)$ is the rightmost vertex before u in π if $u \notin N(w)$. Let l' be the leaf immediately after l_v. Then, for all leaves l between l' and l_u, including l' and l_u, the values of $c_v^{\text{rev}}(l)$ increase by $\delta = c_{vw} - c_{wv}$. To reflect such changes efficiently, instead of adding δ to the $\gamma(l)$ of each leaf l, we add δ to the $\gamma(x)$ of inner nodes x as close as possible to r_w such that all leaves in $L(x)$ are between l' and l_u. An example of this update is shown in Fig. 8. We also update the position of the l_v, taking it from its original position and adding it to the right of l_u.

The update of the tree for each w is executed through the following steps. Note that in these steps and throughout the remainder of this subsection unless otherwise stated, π is the permutation before changing the position of v.

1. Find the leaf l_v by setting $x := r_w$ and repeat the following: if $\pi^{-1}(v_{\text{name}}(x)) < \pi^{-1}(v)$, set x to its right sibling; otherwise, set x to its left child unless x is a leaf, setting $l_v := x$ in case x is a leaf. (We keep this $x(= l_v)$ for later computation.)
2. Find the leaf l_u, where l_u is the rightmost leaf with $\pi^{-1}(v_{\text{name}}(l_u)) \le \pi^{-1}(u)$. This can be done by using a procedure similar to the one used in the previous step. If $l_v = l_u$, stop (in this case, no update is necessary on this tree).
3. Add a leaf y to the right of l_u, and set $v_{\text{name}}(y) := v$ and $\gamma(y) := \gamma_{\min}(y) := \gamma_{\min}(l_u)$. (Inner nodes may be created according to the insertion operation for 2–3 trees.)

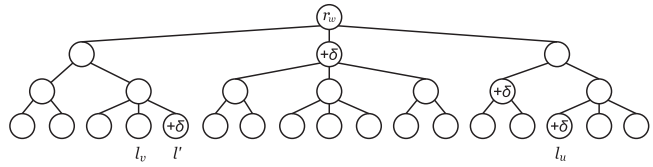


Fig. 8. For all leaves l between leaves l' and l_u, the values of $\gamma(P(r_w, l))$ are modified by adding δ to the nodes labeled with "$+\delta$".

4. While x has a right sibling different from y, repeat the following: set x to its right sibling and then add $\delta = c_{vw} - c_{wv}$ to $\gamma(x)$ and to $\gamma_{\min}(x)$. Update $\gamma_{\min}(p(x))$, where $p(x)$ denotes the parent of node x.
5. While y has a left sibling different from x, repeat the following: set y to its left sibling and then add δ to $\gamma(y)$ and to $\gamma_{\min}(y)$. Update $\gamma_{\min}(p(y))$.
6. Set $x := p(x)$ and $y := p(y)$. If $x \neq y$, return to Step 4; otherwise, update the values in the ancestors of x if necessary.
7. Delete l_v from the tree. (Inner nodes may be removed according to the deletion operation for 2–3 trees.)

This update procedure can be done in $O(\log d_w)$ time for each $w \in N(v)$, and hence it takes $\sum_{w \in N(v)} O(\log d_v) = O(\Delta \log \Delta)$ time to update all the necessary trees.

Let $B(v, u)$ be the set of vertices in $N(v)$ whose positions in π are between the vertices v and u, i.e., $B(v, u) = \{w \in N(v) | \pi^{-1}(v) \leq \pi^{-1}(w) \leq \pi^{-1}(u)\}$. We also have to update $cost(\pi)$, $cost(v, \pi)$ and $cost(w, \pi)$ for all vertices $w \in B(v, u)$. The values of $cost(\pi)$ and $cost(v, \pi)$ are updated by subtracting $\sum_{w \in B(v,u)} (c_{wv} - c_{vw})$ from both of them. To update the costs of the other vertices w, we subtract $c_{wv} - c_{vw}$ from $cost(w, \pi)$ for each $w \in B(v, u)$. This cost update can be done in $O(d_v)$ time.

The case with $\pi^{-1}(v) > \pi^{-1}(u)$ is symmetric, and the above procedures are slightly modified as follows. The changes in the procedure to update each tree are: in Step 3, we modify the rule to initialize $\gamma(y)$ and $\gamma_{\min}(y)$ as $\gamma(y) := \gamma(l_u) + \delta$ and $\gamma_{\min}(y) := \gamma_{\min}(l_u) + \delta$; in Steps 4 and 5, we exchange left and right. In addition, we update the values of $cost(\pi)$, $cost(v, \pi)$ and $cost(w, \pi)$ for all vertices $B(u, v) = \{w \in N(v) | \pi^{-1}(u) \leq \pi^{-1}(w) \leq \pi^{-1}(v)\}$ by subtracting $\sum_{w \in B(u,v)} (c_{vw} - c_{wv})$ from $cost(\pi)$ and from $cost(v, \pi)$, and $c_{vw} - c_{wv}$ from $cost(w, \pi)$ for each $w \in B(u, v)$.

After updating the trees and the costs of the vertices and of the solution, we update the arrays π and π^{-1}. This update can be done in $O(n)$ time.

Figure 9 shows the updates on the tree of v_2 represented in Fig. 7, with the new solution π' obtained by inserting v_1 after v_9. In this case, $c_{v_1 v_2} - c_{v_2 v_1} = -35$ and $cost(v_2, \pi') = 50$.

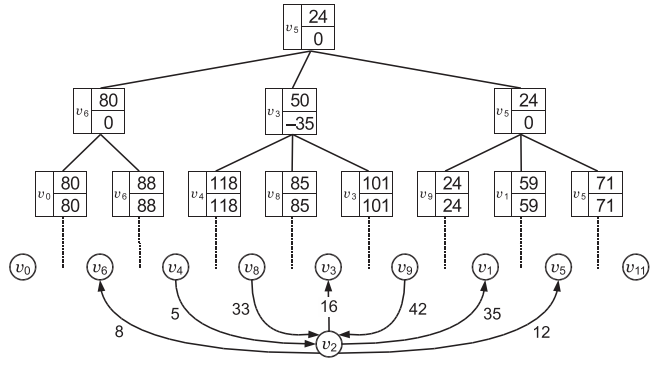


Fig. 9. Updated tree for v_2.

Based on the analysis presented above, we can state the following:

Theorem 2. *The one-round time for the TREE algorithm is* $O(n + \Delta \log \Delta)$. *The data structure of TREE for a given permutation can be built from scratch in* $O(m)$ *time.*

Note that because $\Delta = O(n)$, the one-round time of the TREE algorithm is asymptotically faster than that of SCST. Further discussion about the one-round time of the TREE algorithm is given in Appendix A. Algorithm 3 shows the framework of the TREE algorithm.

Algorithm 3. TREE algorithm
input: graph G and an initial solution π_{init}
output: a locally optimal solution π and its cost $cost(\pi)$
$\pi \leftarrow \pi_{\text{init}}$, $cost \leftarrow \infty$
create the lists of $cost(v, \pi)$ and trees for all vertices v as described in Section 3.1 ($O(m)$)
while $cost(\pi) < cost$ **do**
  $cost \leftarrow cost(\pi)$
  search the roots of the trees to find a vertex v with $\gamma_{\min}(r_v) < cost(v, \pi)$ ($O(n)$)
  if such a vertex v is found **then**
    search the tree of v for vertex u after which v should be inserted ($O(\log d_v)$)
    update the trees of all vertices $w \in N(v)$ ($O(\Delta \log \Delta)$)
    update the value of $cost(\pi)$ and $cost(v, \pi)$ ($O(d_v)$)
    update the value of $cost(w, \pi)$ for all vertices $w \in B(v, u)$ ($O(d_v)$)
    update π ($O(n)$)
  end if
end while
output π and $cost$

4. Computational results

To evaluate the algorithms presented in this paper, we compared their performance with the local search[2] method proposed in Schiavinotto and Stützle (2004), which is referred to as SCST. This method is known as the most efficient one presented so far with respect to the one-round time of the local search using the insert neighborhood, and we implemented it respecting the suggested one-round running time of $O(n^2)$. The codes were written in C language and all the algorithms were run on a PC with an Intel Xeon (NetBurst) 3.0 GHz processor and 8GB RAM.

To compare the results obtained by our algorithms with the ones presented in the literature, throughout Section 4 and Appendix B, we report the objective values in terms of the sum of the elements in the upper triangle (excluding values in the diagonal) of the matrix formulation of the LOP, because the values reported in other papers use this objective function.

[2]Here, we used the word "local search" to mean the basic framework described as Algorithm 1.

Table 1
Average one-round time (in seconds) of the algorithms using instances from the literature

Instance type	#instances	n	SCST	LIST	TREE
LOLIB	49	44–60	0.000014448	0.000027259	0.000036364
SGB	25	75	0.000036076	0.000066765	0.000045455
Random I	25	100	0.000067580	0.000187785	0.000099991
Random I	25	150	0.000142201	0.000652477	0.000189626
Random I	25	200	0.000249385	0.001222563	0.000403511
Random II	25	100	0.000066083	0.000145406	0.000082560
Random II	25	150	0.000143814	0.000661313	0.000178351
Random II	25	200	0.000252608	0.001102779	0.000370730

4.1. Preliminary experiments

We first evaluated the algorithms using representative benchmark instance sets named LOLIB, SGB, Random Type I and II, which are available at Raphael Martí's web site;[3] their recent information is summarized in Garcia et al. (2006). Among these instances, optimal solutions were reported only for the Random Type I instances with $n = 35$, which are not evaluated in this paper due to their small size, and the ones in the LOLIB library. Using CPLEX 10.0 with the integer programming formulation presented in Section 1, we could find optimal solutions for all the instances from the SGB library, which have $n = 75$. In fact, we found that the solutions for these instances reported by Garcia et al. (2006) in their appendix were optimal.

For each instance, we performed a local search using SCST, LIST and TREE once, starting from a same random solution, until a locally optimal solution was found. All the algorithms used the best move strategy. Table 1 shows the average one-round times of all the instances of each set, separated by type and size (number of vertices n).

Although LIST and TREE have better worst-case time complexities than SCST, we can observe from the table that the average one-round time of SCST is smaller than both algorithms for all the instance sets. TREE is faster than LIST for all instances except the LOLIB ones, and the difference between TREE and SCST is not large. These results can be explained by the fact that the proposed algorithms are more complex than SCST; this increases their constant factors in the time complexity and makes them more suitable to instances of larger size, in which the influence of constant factors is smaller.

It should be noted that algorithms SCST and LIST were implemented so that when they start from a same initial solution and use the same move strategy, they move to the same neighbor solution in each step of the algorithm, finding exactly the same locally optimal solution. However, the TREE algorithm may find different solutions since the trajectory taken by the search is not necessarily the same, a difference caused by the inherent difference in the data structure of the algorithm (e.g., when there is more than one neighbor solution with the same cost value, the criterion used to choose the solution to which to move is different).

[3]http://www.uv.es/~rmarti/paper/lop.html

To certify that the above-mentioned difference in the search trajectory does not cause significant difference in the quality of the locally optimal solutions obtained by SCST (or LIST) and TREE, we evaluated these solutions by comparing them with the best solutions known so far for these instances. Table 2 shows the quality of the solutions found by the algorithms and the time until they reached a locally optimal solution in the same experiment.

The first column contains the instance types. The next three columns show the time (in seconds) necessary for each algorithm to find a locally optimal solution, and the following two columns represent the relative difference between the value of the upper triangle of the optimal or best known solution ($upptri_{\mathrm{opt/best}}$) and the one found by the algorithms ($upptri_{\mathrm{alg}}$), calculated by the formula $(upptri_{\mathrm{opt/best}}-upptri_{\mathrm{alg}})/upptri_{\mathrm{opt/best}}$. The values in the table correspond to the average among all the instances of a certain type.

Although the search trajectory of the TREE algorithm is different from the trajectory of the SCST and LIST algorithms, there is not much difference in the quality of the obtained locally optimal solutions. Besides, even though we are comparing solutions found by one run of local search with solutions obtained by more sophisticated metaheuristics, we can observe that the solutions found by our algorithms and by SCST are quite close to the best ones known so far in terms of quality.

The average times necessary for each algorithm to reach a locally optimal solution presented in Table 2 are almost proportional to the one-round times in Table 1, and they present the same tendencies observed in that table. From these two tables, we can observe that although LIST and TREE are algorithms designed to deal with large instances, they also have reasonably good performance for small instances.

4.2. Experiments with large instances

To evaluate the performance of the algorithms on large instances, we generated random instances of sizes between 500 and 8000 using five values of density (probability that an edge between any two vertices exists) from 1% to 100%. 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,[4] resulting in a total of 150 problem instances.[5] The CPLEX was not able to obtain exact optimal solutions or upper bounds by LP relaxation for these instances due to memory limitations.

Table 3 presents the average one-round time in seconds of the algorithms adopting the best move strategy. The search was conducted starting from a same random solution until a locally optimal solution was found or until the search time exceeded 12 hours.

As expected, although the results of SCST do not fluctuate with density, the values for LIST change significantly. LIST is faster than SCST for sparse instances, with densities between 1 and 10%, being more than 40 times faster than SCST for instances with density 1%. However, for the instances with density 50% and 100%, LIST takes around twice and five times the time of SCST,

[4]http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html

[5]These instances are available at http://www.al.cm.is.nagoya-u.ac.jp/~yagiura/lop/

Table 2
Quality of the solutions found by the proposed algorithms

Instance type	Time (s)			Solution quality		Opt./best
	SCST	LIST	TREE	SCST/LIST	TREE	
LOLIB	0.000612	0.001143	0.001428	0.001859	0.001852	Opt.
SGB	0.002480	0.004599	0.004600	0.003038	0.003038	Opt.
Random I	0.008079	0.022516	0.010679	0.005318	0.005702	Best
Random I	0.026916	0.123501	0.035035	0.004715	0.004722	Best
Random I	0.065910	0.323511	0.106144	0.004638	0.004518	Best
Random II	0.007239	0.015957	0.009679	0.000385	0.000406	Best
Random II	0.026596	0.122341	0.032755	0.000235	0.000230	Best
Random II	0.063471	0.277198	0.093626	0.000233	0.000219	Best

Table 3
Average one-round time (in seconds) of the three algorithms for instances of larger size

n	Algorithm	Density (%)				
		1	5	10	50	100
500	SCST	0.0031065	0.002857	0.002884	0.00268	0.0028
	LIST	0.0000596	0.000861	0.001712	0.01353	0.0404
	TREE	0.0000089	0.000025	0.000065	0.00069	0.0015
1000	SCST	0.0362790	0.035507	0.035307	0.03480	0.0347
	LIST	0.0002347	0.001156	0.008193	0.08093	0.1900
	TREE	0.0000172	0.000095	0.000246	0.00189	0.0038
2000	SCST	0.1891338	0.185940	0.184564	0.18215	0.1806
	LIST	0.0010445	0.024276	0.056051	0.36063	0.8599
	TREE	0.0000465	0.000282	0.000659	0.00457	0.0087
3000	SCST	0.4466199	0.438401	0.435690	0.42880	0.4271
	LIST	0.0075084	0.063650	0.135055	0.85936	2.1023
	TREE	0.0000837	0.000499	0.001130	0.00737	0.0143
4000	SCST	0.8371016	0.830644	0.823773	0.88981	0.8872
	LIST	0.0195729	0.119236	0.251957	1.60380	4.1768
	TREE	0.0001261	0.000723	0.001666	0.01042	0.0200
8000	SCST	4.7873324	4.720022	4.735034	4.75155	4.8053
	LIST	0.0945847	0.517447	1.143853	8.79445	22.6167
	TREE	0.0003397	0.001824	0.004011	0.02413	0.0466

respectively. In the case of dense instances, $m = \Omega(n^2)$ and the worst-case one-round time of LIST becomes $O(n^2)$, which is the same as SCST. Because the data structure used by LIST is more complex, the one-round time of LIST becomes larger than that of SCST in such cases.

TREE shows the best performance, presenting the smallest one-round time among the three methods for any instance class and being more than a hundred times faster than the other methods for large instances.

Another evaluation was done in terms of how much time the algorithms take to find a locally optimal solution. In this evaluation, besides the best move strategy, we used the first admissible

move strategy for SCST and LIST. The results are shown in Table 4. The results correspond to times computed from the beginning of the search, i.e., without considering the times to read instances and build data structures.

The values represent the average amount of time in seconds spent until the algorithms stopped after finding locally optimal solutions, starting from a randomly generated solution. In the table, "T.O." means that the algorithm ran for 12 hours without finding a locally optimal solution for all the instances of that class.

From the results in Table 4, we can see that the TREE algorithm is the only one that could reach a locally optimal solution in less than 12 hours for all the problem classes; TREE reached a

Table 4
Time to reach a locally optimal solution

n	Algorithm	Move strategy	Density (%)				
			1	5	10	50	100
500	SCST	Best	0.9069	1.4504	1.6432	2.0189	2.1747
		First	0.3697	0.8009	0.9705	1.4496	1.6392
	LIST	Best	0.0174	0.4383	0.9751	10.1641	31.8788
		First	0.0016	0.0220	0.0952	19.0473	92.7353
	TREE	Best	0.0026	0.0126	0.0356	0.5183	1.2052
1000	SCST	Best	27.4378	41.2255	47.4462	59.5248	66.6577
		First	6.6118	11.9438	15.2223	21.3865	24.3017
	LIST	Best	0.1776	1.3428	11.0109	138.4430	365.5362
		First	0.0106	0.2212	2.6592	265.2471	1521.3867
	TREE	Best	0.0130	0.1084	0.3172	3.2533	6.8180
2000	SCST	Best	351.9283	496.3623	567.0452	698.2716	756.9867
		First	80.2860	131.8742	155.7777	220.4153	243.8055
	LIST	Best	1.9439	64.8015	172.2008	1382.6318	3603.7983
		First	0.1038	9.4876	61.5922	3729.2617	21018.6371
	TREE	Best	0.0866	0.7513	1.9749	17.1772	35.4242
3000	SCST	Best	1395.6812	1894.9275	2121.5547	2673.6799	2883.9076
		First	330.3462	519.0211	607.3933	825.4955	912.4719
	LIST	Best	23.4644	275.1180	657.6628	5358.4320	14201.3319
		First	0.6265	48.3467	281.6440	16540.8074	T.O.
	TREE	Best	0.2632	2.1393	5.6573	45.2669	92.8771
4000	SCST	Best	3631.5215	5016.5952	5465.2006	6712.5637	7129.9031
		First	965.4084	1456.2072	1737.1501	2308.0524	2548.0524
	LIST	Best	86.4585	738.4881	1720.6672	13585.8200	37759.4419
		First	2.2847	140.4271	812.2275	T.O.	T.O.
	TREE	Best	0.5585	4.3577	11.4467	88.6669	179.2260
8000	SCST	Best	38321.0989	T.O.	T.O.	T.O.	T.O.
		First	20642.0433	24213.2550	28582.3740	(41490.6257)	T.O.
	LIST	Best	963.1642	7139.7846	17717.8765	T.O.	T.O.
		First	31.1471	1750.9126	10232.6312	T.O.	T.O.
	TREE	Best	3.4837	25.2932	62.5775	457.9856	941.0565

Note: The value in parentheses represents the average excluding instance n8000d050-4 for which the time limit was reached without confirming local optimality. See Appendix B for the description of instance names.

locally optimal solution in less than 20 minutes for all instances. The results are similar to the ones of Table 3, with LIST reaching the locally optimal solution faster than SCST for instances with density smaller than 50%, and TREE reaching the locally optimal solution faster than the other two algorithms for almost all the instance classes.

A comparison between the strategies used with the SCST algorithm shows that the algorithm with the first admissible move strategy reaches the locally optimal solution faster than the one with the best move strategy for all instance classes. This result can be explained analyzing the way the algorithm works: using the first admissible move strategy, the average time to find the first improved solution and to update the data structure is much smaller than $O(n^2)$.

The LIST algorithm reached the locally optimal solution faster using the first admissible move strategy than the best move strategy only for the sparse instance classes, with densities up to 10%. For these instances, the update of the data structure can be done much faster than $O(m)$ time, because we only update the lists of the vertices adjacent to the inserted one, and the search time has a bigger influence on the one-round time than the time to update data structures. When making a move using the first admissible move strategy, the mean search time becomes much smaller than $O(m)$, the search time for the best move strategy, while the time to update the data structure does not change significantly, justifying a claim of better performance compared with the best move strategy. For dense instances, although the search time is reduced in the same way as for the sparse ones, the update of the data structure has a greater influence on the one-round time. Moreover, when the first admissible move strategy is adopted, the number of moves needed to reach the locally optimal solution usually becomes significantly larger than the case in which the best move strategy is adopted. For these reasons, the best move strategy has a better performance than the first admissible move strategy for dense instances.

We also checked the quality of the solutions found by the algorithms to certify that there is no significant difference caused by the difference in the search trajectories. Let $upptri_{\mathrm{best}}$ be the sum of the elements in the upper triangle of the best solution obtained among the algorithms. We denote the relative difference of a solution as the percent difference between the value of the solution's upper triangle ($upptri_{\mathrm{alg}}$) and $upptri_{\mathrm{best}}$, given by the expression $((upptri_{\mathrm{best}}-upptri_{\mathrm{alg}})/upptri_{\mathrm{best}})$.

Table 5 shows a comparison between the quality of the solutions found by the algorithms in terms of the average relative difference. A value in the table is the average of five instances with the same size and density. For instances with sizes up to $n = 2000$, all the algorithms reached a locally optimal solution within the search time limit of 12 hours. For these instances, SCST and LIST output exactly the same solutions, because their search trajectories and initial solutions are the same. Hence, their results are shown in the same rows. For larger instances, however, there are cases where SCST and/or LIST did not find a locally optimal solution within the time limit. In such cases, we calculate the average relative error of the best solutions obtained before the time limit, where such values in the table are marked with "†". The quality of the locally optimal solutions found by the algorithms is very similar. The average relative difference calculated for the cases where all the algorithms reached a locally optimal solution was around 0.11%, with the maximum difference being 1.35%. The values of the best solution obtained for each instance (i.e., $upptri_{\mathrm{best}}$) are reported in Appendix B.

Table 5
Comparison between the solution quality of the algorithms for large instances

n	Algorithm	Move strategy	Density (%)				
			1	5	10	50	100
500	SCST/	Best	0.0054982	0.0025412	0.0025618	0.00155345	0.0002045
	LIST	First	0.0059257	0.0029843	0.0003536	0.00101174	0.0000795
	TREE	Best	0.0045777	0.0019622	0.0019442	0.00144725	0.0003182
1000	SCST/	Best	0.0031355	0.0023542	0.0020110	0.00069932	0.0001765
	LIST	First	0.0031153	0	0.0001037	0.00017910	0.0001783
	TREE	Best	0.0045777	0.0019622	0.0019442	0.00144724	0.0003182
2000	SCST/	Best	0.0013769	0.0012828	0.0007816	0.00056895	0.0000896
	LIST	First	0.0017806	0.0000274	0.0001118	0.00019753	0.0000205
	TREE	Best	0.0001872	0.0009571	0.0008445	0.00078218	0.0001828
3000	SCST	Best	0.0018676	0.0012505	0.0012984	0.0001993	0.0000287
		First	0.0009637	0	0.0000093	0.0000905	0.0000571
	LIST	Best	0.0018676	0.0012505	0.0012984	0.0001993	0.0000287
		First	0.0009637	0	0.0000093	0.0000905	0.0092698†
	TREE	Best	0.0005802	0.0012603	0.0006270	0.0002198	0.0001104
4000	SCST	Best	0.0023881	0.0008019	0.0006847	0.0002877	0.0000651
		First	0.0000534	0	0	0	0.0000052
	LIST	Best	0.0023881	0.0008019	0.0006847	0.0002877	0.0000651
		First	0.0000534	0	0	0.0043045†	0.0131117†
	TREE	Best	0.0015005	0.0010240	0.0005836	0.0002424	0.0000917
8000	SCST	Best	0.0020490	0.0021341†	0.0019128†	0.0009763†	0.0002939†
		First	0	0	0	0.0005514†	0.0010877†
	LIST	Best	0.0020490	0.0007573	0.0004246	0.0061765†	0.0055460†
		First	0	0	0	0.0336760†	0.0111911†
	TREE	Best	0.0011449	0.0003593	0.0005003	0.0000902	0

Note: Values marked with "†" were calculated from solutions obtained when the algorithms reached the time limit for at least one instance in that class.

5. Conclusions

In this paper, we studied local search algorithms for the LOP, developing two algorithms, LIST and TREE, that perform the search through the neighborhood of the insert operation efficiently.

The data structure of the LIST algorithm uses $O(m)$ memory space, and its one-round time is $O(m)$. Computational experiments with random instances showed good results for sparse instances with density up to 10% when compared with the results of the $O(n^2)$ time algorithm presented in the literature. LIST does not present significant difficulties in its implementation and can be easily integrated as a part of metaheuristic algorithms, and is strongly recommended for sparse instances.

The TREE algorithm presented the best performance for the search through the whole neighborhood. It utilizes $O(m)$ memory space, and its one-round time is $O(n + \Delta \log \Delta)$. Experiments showed that for all large instances, TREE is the fastest among the algorithms studied in this paper and the only one capable of handling large instances in reasonable time. This algorithm can be incorporated in various heuristic algorithms that are based on local search with the insert neighborhood to make them run faster for large-scale instances.

Acknowledgements

The authors are grateful to Professor Takao Ono for his valuable comments on implementation issues and to Professor Yoshitsugu Yamamoto for his helpful comments about the IP formulation of the LOP. 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

Aho, A.V., Hopcroft, J.E., Ullman, J.D., 1974. *The Design and Analysis of Computer Algorithms*. Addison Wesley, Boston, MA.

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

Charon, I., Hudry, O., 2007. A survey on the linear ordering problem for weighted or unweighted tournaments. *4OR: A Quarterly Journal of Operations Research* 5, 5–60.

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

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

Garey, M.R., Johnson, D.S., 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co Ltd, New York, NY.

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

Grötschel, M., Jünger, M., Reinelt, G., 1985. Facets of the linear ordering polytope. *Mathematical Programming* 33, 43–60.

Huang, G., Lim, A., 2003. Designing a hybrid genetic algorithm for the linear ordering problem. Proceedings of the Genetic and Evolutionary Computation – GECCO 2003, pp. 1053–1064.

Karp, R.M., 1972. Reducibility among combinatorial problems. In Miller, R.E., Thatcher, J.W. (eds) *Complexity of Computer Computations*. Plenum Press, New York, NY, pp. 85–103.

Laguna, M., Marti, R., Campos, V., 1999. Intensification and diversification with elite tabu search solutions for the linear ordering problem. *Computers and Operations Research* 26, 1217–1230.

Schiavinotto, T., Stützle, T., 2003. Search space analysis of the linear ordering problem. In Raidl, G. R., et al. (eds) *Applications of Evolutionary Computing*. Springer, Heidelberg, pp. 197–204.

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

Yagiura, M., Ibaraki, T., 1999. Analyses on the 2 and 3-flip neighborhoods for the max sat. *Journal of Combinatorial Optimization* 3, 95–114.

Appendix A: Analysis of the one-round time of the TREE algorithm

We state in Theorem 2 that the one-round time for the TREE algorithm is $O(n + \Delta \log \Delta)$. However, theoretically this time could be reduced to $O(\Delta \log n)$ using a real RAM model.

To achieve this order of time, we modify our algorithm in two places. The first change is in the search for the vertex to be inserted, which by the current algorithm takes $O(n)$ time. If we keep the values of $\gamma_{\min}(r_v) - cost(v, \pi)$ in a heap with the minimum value in the root, we can

find the smallest difference in constant order of time. To update this heap after the insertion of a vertex v, we have to update the values of d_v nodes of the heap, which takes $O(\Delta \log n)$ time.

The second modification in the algorithm concerns the arrays representing π and π^{-1}, whose update after an insertion by the current algorithm takes $O(n)$ time. We keep π as a linked list so that its update can be done in constant order of time. We then substitute π^{-1} by an array with a function $f : V \to \mathbb{R}$ such that $\pi^{-1}(i) < \pi^{-1}(j) \Leftrightarrow f(i) < f(j)$ for all $i \neq j$. This array is initialized with $f(i) := M\pi^{-1}(i), \forall i$ and updated in constant order of time as follows: when the vertex in position i is inserted between positions j and j', let $f(i) := (f(j) + f(j'))/2$.

However, to realize such changes on real computers, we should consider implementation issues about the number of digits to express the values of f. One way of expressing it is to define f to be a mapping from V to integers $\{1, 2, \ldots, nM\}$ (in fact, we have to keep dummies at both ends, having $(n+1)M$ instead of nM, but we ignored this for simplicity). We initialize f with $f(\pi(i)) := iM$, and update it with $f(i) := \lfloor (f(j) + f(j'))/2 \rfloor$. The memory space necessary to keep f is $O(\log(nM))$ bits.

By expressing f this way, each time a vertex is inserted after a position j, the space between $f(j)$ and $f(j+1)$ is reduced by half. After $\log M$ insertions after the same position j, the difference between $f(j)$ and $f(j+1)$ is reduced to one. In this case, to update f it is necessary to initialize it again using π. Let R be the number of moves executed before such a reset of f.

Implementing these modifications in the algorithm, the amortized one-round time becomes as follows:

$$O\left(\Delta \log n + \log(nM) + \Delta(\log \Delta)(\log(nM)) + \frac{n \log(nM)}{R}\right), \tag{A1}$$

where the first term corresponds to the update of the heap that keeps the values of $\gamma_{\min}(r_v) - cost(v, \pi)$; the second one corresponds to the time to update f; the third one corresponds to the time to update the trees (it is necessary to look at the values of f when looking for a leaf of the tree); and the last one corresponds to the time to reinitialize f after R insertions, supposing that after R insertions we had $\log M$ insertions after the same position.

Supposing the worst-case scenario where all the insertions are made after the same vertex, we would have $R = \log M$ in the last term of the expression. Then, the last term becomes greater than n, not improving the time complexity of the original algorithm. However, as the denominator represents a very pessimistic pathological worst-case scenario, the one-round time of our algorithm can be reduced if a better approximation for it can be found.

Appendix B: Best solutions obtained for large instances

Tables 6–11 show the best solutions obtained by the local search algorithms for large instances. The codes were written in C language and all the algorithms were run on a PC with an Intel Xeon (NetBurst) 3.0 GHz processor and 8GB RAM. The values of the solutions are given by the sum of the elements in the upper triangle (excluding values in the diagonal) of the matrix formulation of the LOP. Instances are named as follows: n[*number of vertices*]d[*density*] – [*instance ID*]. Each row of the tables contains the instance name, the value of the best solution obtained, the name and the

strategy used by the algorithm that obtained that solution and the time in seconds that the algorithm needed to find that solution. In case of ties, the algorithm with the smaller time is reported.

Table 6
Best solutions found by the algorithms for instances with $n = 500$

Name	Upper triangle	Algorithm	Strategy	Time (s)
n0500d001-1	58886	LIST	Best	0.0180
n0500d001-2	52794	LIST	First	0.0020
n0500d001-3	54170	LIST	First	0.0010
n0500d001-4	59191	TREE	Best	0.0030
n0500d001-5	58429	LIST	First	0.0020
n0500d005-1	234995	LIST	Best	0.8409
n0500d005-2	234053	LIST	First	0.0210
n0500d005-3	237689	TREE	Best	0.0130
n0500d005-4	238056	LIST	First	0.0220
n0500d005-5	234934	TREE	Best	0.0120
n0500d010-1	434925	LIST	First	0.1030
n0500d010-2	433628	LIST	First	0.0800
n0500d010-3	433107	LIST	First	0.0900
n0500d010-4	435776	TREE	Best	0.0380
n0500d010-5	434216	LIST	Best	0.7309
n0500d050-1	2097339	SCST	First	1.4238
n0500d050-2	2095533	SCST	First	1.4348
n0500d050-3	2087570	SCST	Best	2.1307
n0500d050-4	2091577	SCST	First	1.4528
n0500d050-5	2098933	TREE	Best	0.5489
n0500d100-1	6514410	SCST	Best	2.1047
n0500d100-2	6507318	SCST	First	1.5948
n0500d100-3	6524477	TREE	Best	1.3068
n0500d100-4	6519873	SCST	First	1.5968
n0500d100-5	6514317	SCST	Best	2.3546

Table 7
Best solutions found by the algorithms for instances with $n = 1000$

Name	Upper triangle	Algorithm	Strategy	Time (s)
n1000d001-1	209184	TREE	Best	0.0140
n1000d001-2	204955	TREE	Best	0.0130
n1000d001-3	203901	LIST	First	0.0110
n1000d001-4	212440	TREE	Best	0.0130
n1000d001-5	210741	LIST	Best	0.1700
n1000d005-1	856346	LIST	First	0.2390
n1000d005-2	860259	LIST	First	0.2050
n1000d005-3	864470	LIST	First	0.2290
n1000d005-4	863149	LIST	First	0.2110
n1000d005-5	859514	LIST	First	0.2220
n1000d010-1	1607713	LIST	First	2.5586
n1000d010-2	1623827	LIST	Best	11.0643
n1000d010-3	1626046	LIST	First	2.7226
n1000d010-4	1623101	LIST	First	2.6086
n1000d010-5	1620999	LIST	First	2.7516
n1000d050-1	8079273	SCST	First	21.0258
n1000d050-2	8066525	SCST	First	20.9078
n1000d050-3	8071592	SCST	First	21.6787
n1000d050-4	8070244	TREE	Best	3.3035
n1000d050-5	8066843	TREE	Best	3.1835
n1000d100-1	25764671	SCST	Best	71.4971
n1000d100-2	25762629	SCST	Best	67.6057
n1000d100-3	25799768	SCST	First	24.3413
n1000d100-4	25816512	SCST	Best	70.0514
n1000d100-5	25763346	SCST	First	24.0143

Table 8
Best solutions found by the algorithms for instances with $n = 2000$

Name	Upper triangle	Algorithm	Strategy	Time (s)
n2000d001-1	768991	LIST	First	0.1050
n2000d001-2	774491	TREE	Best	0.0900
n2000d001-3	747027	TREE	Best	0.0830
n2000d001-4	771736	LIST	Best	1.9327
n2000d001-5	768251	TREE	Best	0.0850
n2000d005-1	3204926	LIST	First	9.4356
n2000d005-2	3239560	LIST	First	9.8305
n2000d005-3	3196448	LIST	First	9.0626
n2000d005-4	3231299	TREE	Best	0.7929
n2000d005-5	3199913	LIST	First	9.4386
n2000d010-1	6108739	LIST	First	60.8807
n2000d010-2	6140377	LIST	First	62.7295
n2000d010-3	6079840	LIST	Best	178.8318
n2000d010-4	6134018	LIST	First	62.0506
n2000d010-5	6123804	LIST	First	62.0056
n2000d050-1	31463156	SCST	First	221.0604
n2000d050-2	31484072	SCST	First	220.2955
n2000d050-3	31455461	SCST	First	221.9873
n2000d050-4	31475543	SCST	Best	743.0380
n2000d050-5	31458031	TREE	Best	17.6003
n2000d100-1	102241830	TREE	Best	38.8861
n2000d100-2	102224905	SCST	First	248.6442
n2000d100-3	102273950	SCST	First	236.2331
n2000d100-4	102237648	SCST	Best	755.0732
n2000d100-5	102214090	SCST	First	240.8864

Table 9
Best solutions found by the algorithms for instances with $n = 3000$

Name	Upper triangle	Algorithm	Strategy	Time (s)
n3000d001-1	1640816	TREE	Best	0.2580
n3000d001-2	1653347	LIST	First	0.6439
n3000d001-3	1618673	TREE	Best	0.2560
n3000d001-4	1647727	TREE	Best	0.2610
n3000d001-5	1657883	TREE	Best	0.2700
n3000d005-1	6988497	LIST	First	50.3943
n3000d005-2	7006180	LIST	First	48.3736
n3000d005-3	6928462	LIST	First	47.0029
n3000d005-4	6996155	LIST	First	47.2898
n3000d005-5	6989847	LIST	First	48.6726
n3000d010-1	13374308	TREE	Best	5.5632
n3000d010-2	13421082	LIST	First	280.4454
n3000d010-3	13365619	LIST	First	280.4754
n3000d010-4	13422891	LIST	First	281.2582
n3000d010-5	13433851	LIST	First	286.9524
n3000d050-1	69857962	SCST	First	807.9652
n3000d050-2	69977439	SCST	Best	2682.0143
n3000d050-3	69943319	SCST	First	839.4264
n3000d050-4	70006796	TREE	Best	45.9660
n3000d050-5	69991403	TREE	Best	48.2067
n3000d100-1	229119563	SCST	Best	2871.3665
n3000d100-2	229129631	SCST	First	913.8711
n3000d100-3	229236975	SCST	First	911.5104
n3000d100-4	229146533	SCST	Best	3086.8837
n3000d100-5	229166462	SCST	Best	2908.6718

Table 10
Best solutions found by the algorithms for instances with $n = 4000$

Name	Upper triangle	Algorithm	Strategy	Time (s)
n4000d001-1	2828898	LIST	First	2.2947
n4000d001-2	2828661	LIST	First	2.2587
n4000d001-3	2796290	TREE	Best	0.5599
n4000d001-4	2839726	LIST	First	2.3906
n4000d001-5	2827592	LIST	First	2.2257
n4000d005-1	12116832	LIST	First	140.0337
n4000d005-2	12154754	LIST	First	140.6926
n4000d005-3	12091130	LIST	First	141.7455
n4000d005-4	12144547	LIST	First	141.0466
n4000d005-5	12118053	LIST	First	138.6169
n4000d010-1	23360678	LIST	First	805.4136
n4000d010-2	23390745	LIST	First	810.6238
n4000d010-3	23356065	LIST	First	818.4386
n4000d010-4	23427578	LIST	First	820.9172
n4000d010-5	23409923	LIST	First	805.7445
n4000d050-1	123341557	SCST	First	2264.1448
n4000d050-2	123447039	SCST	First	2257.0479
n4000d050-3	123422850	SCST	First	2393.3452
n4000d050-4	123588652	SCST	First	2339.7823
n4000d050-5	123424670	SCST	First	2285.6885
n4000d100-1	406444129	TREE	Best	172.2638
n4000d100-2	406419735	SCST	Best	7181.0443
n4000d100-3	406592309	SCST	First	2577.4042
n4000d100-4	406504913	SCST	First	2603.4922
n4000d100-5	406432703	SCST	First	2574.0947

Table 11
Best solutions found by the algorithms for instances with $n = 8000$

Name	Upper triangle	Algorithm	Strategy	Time (s)
n8000d001-1	10484599	LIST	First	30.9733
n8000d001-2	10469314	LIST	First	31.2632
n8000d001-3	10420731	LIST	First	31.3552
n8000d001-4	10441469	LIST	First	30.6723
n8000d001-5	10457025	LIST	First	31.4712
n8000d005-1	46327825	LIST	First	1751.0948
n8000d005-2	46341305	LIST	First	1762.1131
n8000d005-3	46224313	LIST	First	1756.7589
n8000d005-4	46254387	LIST	First	1731.8527
n8000d005-5	46263979	LIST	First	1752.7435
n8000d010-1	90261660	LIST	First	10296.5477
n8000d010-2	90415024	LIST	First	10298.5214
n8000d010-3	90252150	LIST	First	10180.8793
n8000d010-4	90365006	LIST	First	10194.0083
n8000d010-5	90414766	LIST	First	10193.1994
n8000d050-1	486602592	SCST	First	41539.9810
n8000d050-2	486601245	TREE	Best	454.8409
n8000d050-3	486669139	SCST	First	42475.9327
n8000d050-4	486627154	TREE	Best	442.9337
n8000d050-5	486453433	SCST	First	41825.7835
n8000d100-1	1618034116	TREE	Best	985.2332
n8000d100-2	1618317029	TREE	Best	930.0766
n8000d100-3	1618255305	TREE	Best	868.8959
n8000d100-4	1618394063	TREE	Best	977.1365
n8000d100-5	1618501857	TREE	Best	943.9405