Vol.2009-AL-127 No.8
2009/11/27

大規模な線形順序付け問題に対する高性能な局所探索アルゴリズム

櫻庭　セルソ智[†1]　　柳　浦　睦　憲[†1]

各辺に重みを持つ n 頂点の有向グラフが与えられたとき，グラフの頂点を一列に並べ，前向きの辺の重みの合計を最小化する問題を線形順序付け問題と呼ぶ．この問題に対し，挿入近傍を用いる局所探索法において近傍探索を高速に行うアルゴリズムを提案する．このアルゴリズムを TREE と呼ぶ．TREE は $O(n + \Delta \log \Delta)$ 時間で近傍全体を探索できる (Δ はグラフの最大次数)．計算実験により，TREE が既存の方法よりも優れた結果を得ることを確認した．

A local search algorithm capable of dealing with large instances of the linear ordering problem

Celso Satoshi Sakuraba[†1] and Mutsunori Yagiura[†1]

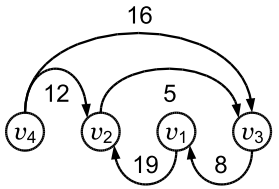
Given a directed graph with n vertices and costs on the edges connecting them, the linear ordering problem consists of finding a permutation of the vertices such that the total cost of the reverse edges is minimized. We present a local search algorithm named TREE for the neighborhood of the *insert* operation, which can search the whole neighborhood in $O(n + \Delta \log \Delta)$ time, where Δ is the maximum degree of the graph. Computational experiments show good results when compared with other methods proposed in the literature.

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 $\pi : \{1, \ldots, n\} \to V$, 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:

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

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 (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 Figure 1. In the figure, each vertex v_i corresponds to column and row i in the matrix.



	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 LOP with cost 19+8=27

The LOP has a number of real world applications in various fields[7], 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

†1 名古屋大学 Nagoya University

© 2009 Information Processing Society of Japan

Vol.2009-AL-127 No.8
2009/11/27

problem[6),9)],[⋆1] the LOP has been vastly studied in the literature since the appearance of the first paper about it[4)], and many exact and heuristic methods have been proposed to solve it. Good literature reviews about the LOP are given by Schiavinotto and Stützle[12)] and Charon and Hudry[3)].

Among the heuristic approaches, there are a number of metaheuristics to handle the LOP, such as tabu search[10)], scatter search[2)], variable neighborhood search[5)] and genetic algorithm[8)]. 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 with no better solution in its neighborhood is called locally optimal.

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*[13)].

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

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*[8)]. As expected, Schiavinotto and Stützle[12)] 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.

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

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:

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

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[12)], is the best algorithm found so far vis-à-vis the one-round time of local search methods for the LOP.

In this paper, we propose an algorithm named TREE for the search through the insert neighborhood. The one-round time of TREE algorithm is $O(n + \Delta \log \Delta)$, 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 results showed that TREE has a good performance when compared to other methods proposed in the literature, being more than a hundred times faster than these methods for large instances.

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

2. 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[1)]. 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.

© 2009 Information Processing Society of Japan

Vol.2009-AL-127 No.8
2009/11/27

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., $\pi_v^{-1}(u) < \pi_v^{-1}(w) \iff \pi^{-1}(u) < \pi^{-1}(w)$ 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 $v \in V$).

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 Figure 2. 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, since 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.

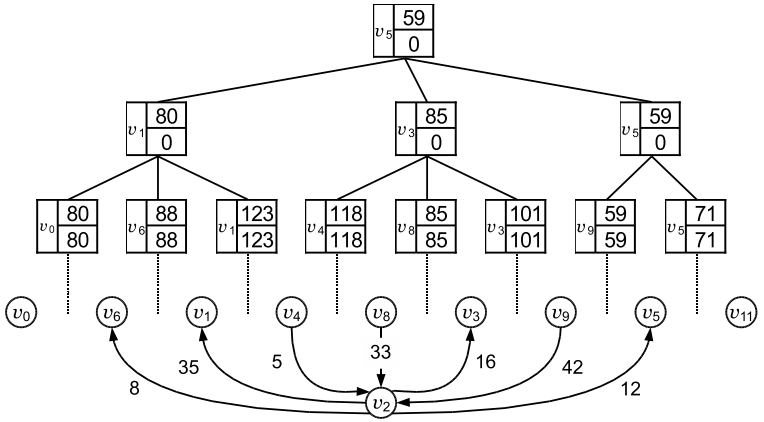


Fig. 2 The vertices adjacent to vertex v_2 and the tree corresponding to v_2

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

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

In Figure 2, $cost(v_2, \pi) = 8 + 35 + 42 = 85$. 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 Figure 2). 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 Figure 2, $v_{\mathrm{name}}(x)$ 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.

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 Figure 2, $\gamma_{\min}(x)$ 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

© 2009 Information Processing Society of Japan

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 $O(\sum_{v\in V}(d_v+1)) = O(m+n) = O(m)$.

2.1 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 Sakuraba and Yagiura[11].

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 $v_{\text{name}}(l) = v_0$ 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 $v_{\text{name}}(l) := u$ 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. Since 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 $cost(v,\pi_{\text{init}})$ 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}.

2.2 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 to 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 $u = v_{\text{name}}(l_{\text{pos}})$.

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 Figure 3. We also update the position of the l_v, taking it from is 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,

© 2009 Information Processing Society of Japan

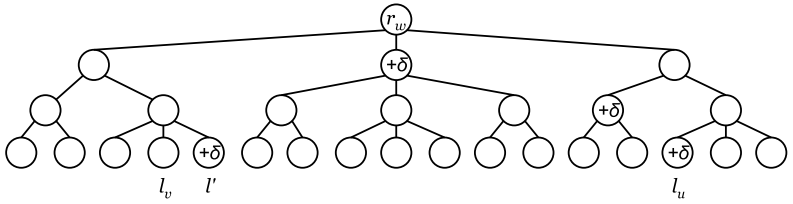


Fig. 3 For all leaves l between leaves l' and l_u, the values of $\gamma(P(r_w, l))$ are modified by adding δ to the nodes labeled with "$+\delta$"

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 $\pi^{-1}(v_{\mathrm{name}}(l_u)) \leq \pi^{-1}(u)$. 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 $v_{\mathrm{name}}(y) := v$ and $\gamma(y) := \gamma_{\min}(y) := \gamma_{\min}(l_u)$. (Inner nodes may be created according to the insertion operation for 2-3 trees.)

(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., $B(v, u) = \{w \in N(v) \mid \pi^{-1}(v) \leq \pi^{-1}(w) \leq \pi^{-1}(u)\}$. 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 $B(u, v) = \{w \in N(v) \mid \pi^{-1}(u) \leq \pi^{-1}(w) \leq \pi^{-1}(v)\}$ 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 4 shows the updates on the tree of v_2 represented in Figure 2, 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 $cost(v_2, \pi') = 50$.

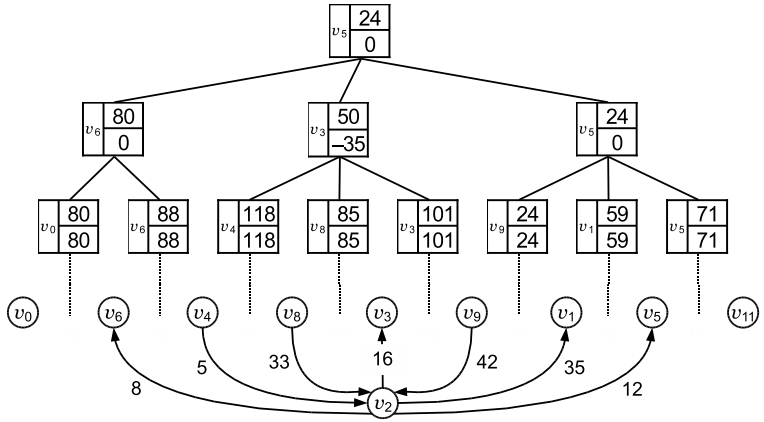


Fig. 4 Updated tree for v_2

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

Theorem. *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 asymp-

© 2009 Information Processing Society of Japan

totically faster than that of SCST.

3. Computational Results

We evaluate the performance of the TREE algorithm using a set of randomly generated instances of sizes (number of vertices) between 500 and 8000. Five values of density (probability of an edge between any two vertices exist) 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[⋆1].

We compare the one-round time of TREE algorithm with the ones obtained by the methods proposed by Schiavinotto and Stützle[12] and Sakuraba and Yagiura[11], which are referred to as SCST and LIST, respectively. The codes were written in the C language and all the algorithms were run on a PC with an Intel Xeon (3.0 GHz) processor and 8GB RAM.

Table 1 presents the average one-round time in seconds of the algorithms. The results of SCST and LIST were taken from Sakuraba and Yagiura[11], where only the better results between them are shown in the table due to space limitations (i.e., for the instances with density up to 10%, LIST was always faster than SCST, and vice versa). The three algorithms adopted the best move strategy, i.e., the algorithm searches through the whole neighborhood and then moves to the neighbor that has the minimum cost.

Table 1 TREE Algorithm Results

dens.	1%		5%		10%		50%		100%	
n	LIST	TREE	LIST	TREE	LIST	TREE	SCST	TREE	SCST	TREE
500	.00006	.00001	.00086	.00003	0.0017	.000065	0.0027	.00069	0.0028	.0015
1000	.00024	.00002	.00116	.00010	0.0082	.000246	0.0348	.00189	0.0347	.0038
2000	.00105	.00005	.02428	.00028	0.0561	.000659	0.1822	.00457	0.1806	.0087
3000	.00751	.00008	.06365	.00050	0.1351	.001130	0.4288	.00737	0.4271	.0143
4000	.01957	.00013	.11924	.00072	0.2520	.001666	0.8898	.01042	0.8872	.0200
8000	.09459	.00034	.51745	.00182	1.1439	.004011	4.7516	.02413	4.8053	.0466

⋆1 http://www.math.sci.hiroshima-u.ac.jp/∼m-mat/MT/ SFMT/index.html

We can conclude from the table that TREE has 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.

4. Conclusions

In this paper, we studied local search algorithms for the LOP and presented an algorithm that can perform the search through the neighborhood of the insert operation efficiently.

The TREE algorithm proposed in this paper utilizes $O(m)$ memory space and its one-round time is $O(n + \Delta \log \Delta)$. Experiments showed that TREE is the fastest among the algorithms studied in this paper for all tested instances.

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) Aho, A.V., Hopcroft, J.E. and Ullman, J.D.: *The Design and Analysis of Computer Algorithms*, Addison Wesley, Boston (1974).
2) Campos, V., Glover, F., Laguna, M. and Martí, R.: An experimental evaluation of a scatter search for the linear ordering problem, *Journal of Global Optimization*, Vol.21, No.4, pp.397–414 (2001).
3) Charon, I. and Hudry,O.: A survey on the linear ordering problem for weighted or unweighted tournaments, *4OR: A Quarterly Journal of Operations Research*, Vol.5, No.1, pp.5–60 (2007).
4) Chenery, H.B. and Watanabe, T.: International comparisons of the structure of production, *Econometrica*, Vol.26, No.4, pp.487–521 (1958).
5) Garcia, C.G., Pérez-Brito, D., Campos, V. and Martí, R.: Variable neighborhood search for the linear ordering problem, *Computers & Operations Research*, Vol.33, No.12, pp.3549–3565 (2006).
6) Garey, M.R. and Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co Ltd, New York (1979).
7) Grötschel, M., Jünger, M. and Reinelt, G.: A cutting plane algorithm for the linear ordering problem, *Operations Research*, Vol.32, No.6, pp.1195–1220 (1984).

© 2009 Information Processing Society of Japan

Vol.2009-AL-127 No.8
2009/11/27

8) Huang, G.F. and Lim, A.: Designing a hybrid genetic algorithm for the linear ordering problem, *GECCO 2003 Proceedings*, Chicago, Illinois, pp.1053–1064 (2003).
9) Karp, R.M.: Reducibility Among Combinatorial Problems, *Complexity of Computer Computations* (Miller, R.E. and Thatcher, J.W.(ed.)), Plenum Press, New York, pp.85–103 (1972).
10) Laguna, M., Martí, R. and Campos, V.: Intensification and diversification with elite tabu search solutions for the linear ordering problem, *Computers & Operations Research*, No.26, No.12, pp.1217–1230 (1999).
11) Sakuraba, C.S. and Yagiura, M.: A local search algorithm efficient for sparse instances of the linear ordering problem, *WAAC08 Proceedings*, Fukuoka, Japan, pp. 44–50 (2008).
12) Schiavinotto, T. and Stützle, T.: The linear ordering problem: Instances, search space analysis and algorithms, *Journal of Mathematical Modelling and Algorithms*, Vol.3, No.4, pp.367–402 (2004).
13) Yagiura, M. and Ibaraki, T.: Analyses on the 2 and 3-Flip Neighborhoods for the MAX SAT, *Journal of Combinatorial Optimization*, Vol.3, No.1, pp.95–114 (1999).

© 2009 Information Processing Society of Japan