

# An Efficient Local Search Algorithm 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,  
sakuraba@al.cm.is.nagoya-u.ac.jp, yagiura@nagoya-u.jp

**Keywords:** Linear ordering problem, local search.

**Abstract.** Given a directed graph with  $n$  vertices,  $m$  edges and costs on the edges, the linear ordering problem consists of finding a permutation  $\pi$  of the vertices so that the total cost of the reverse edges is minimized. We present a local search algorithm for the neighborhood of the *insert* move that performs a search through the neighborhood in  $O(n + \Delta \log \Delta)$  time, where  $\Delta$  represents the maximum vertex degree. Computational experiments show that the proposed algorithm presented the best results compared to other methods 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. For convenience, we assume  $c_{uv} = 0$  for all  $(u, v) \notin E$  and that if we regard  $G$  as an undirected graph, it is connected (which implies  $m \geq n - 1$ , where  $m = |E|$ ). Denoting the permutation by  $\pi : \{1, \dots, n\} \rightarrow V$ , where  $\pi(i) = v$  (equivalently,  $\pi^{-1}(v) = i$ ) signifies that  $v$  is the  $i$ th element of  $\pi$ , the total cost of the reverse edges is formally defined as follows:

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

The LOP has a number of real world applications in various fields (Grötschel et al., 1984), among which the most widely known is the triangularization of economic input-output matrices. Known as an NP-hard problem (Chanas and Kobylański, 1996), the LOP has been vastly studied in the literature. Good literature reviews about the LOP are given by Schiavinotto and Stützle (2004) and Garcia et al. (2006).

Among heuristic approaches, there is a number of metaheuristics to handle the LOP that use local search methods to refine the quality of their solutions. The most widely known local search 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  $j$ .

According to Huang and Lim (2003) and Schiavinotto and Stützle (2004), it is not advantageous to use *interchange* compared to *insert*, and indeed most of the algorithms proposed so far use the *insert* operation.

We denote by *search through the neighborhood* the task of finding an improved solution or concluding that no such solution exists (i.e., the current solution is locally optimal). The

necessary computation time to perform such a task, including the time to update relevant data structures, is called *one-round time*.

A straightforward search through the insert neighborhood can be conducted in  $O(n^3)$  time. By conducting the search in an ordered way, the one-round time can be reduced to  $O(n^2)$ . This result, presented by [Schiavinotto and Stützle \(2004\)](#), is the best one found so far concerning 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  $\Delta$  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.

## 2 The TREE algorithm

The TREE algorithm uses a balanced search tree data structure to make the search through the insert neighborhood efficient. Our implementation is based on the 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. A tree is build 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  $\pi$ .

Let  $N(v)$  be the set of all vertices adjacent to a vertex  $v$  (i.e.,  $N(v) = \{u \in V \mid (u, v) \in E \text{ or } (v, u) \in E\}$ ). We denote by  $\pi_v : \{1, 2, \dots, d_v\} \rightarrow N(v)$  the permutation of the vertices  $u \in N(v)$  having the same order as  $\pi$ , 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  $\pi$  and of each  $\pi_v$  ( $\pi(0) = v_0$  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  $\pi_v$ . 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 1. 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 positions to  $v_2$  have 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  $v$  for the current solution  $\pi$ . This cost corresponds to the sum of the costs of all reverse edges connected with  $v$  for the current solution and is given by

$$\text{cost}(v, \pi) = \sum_{i < \pi^{-1}(v)} c_{v, \pi(i)} + \sum_{i > \pi^{-1}(v)} c_{\pi(i), v}. \quad (2)$$

In Figure 1,  $\text{cost}(v_2, \pi) = 8 + 35 + 42 = 85$ . For the current solution  $\pi$ , we keep a list of  $\text{cost}(v, \pi)$  for all the vertices  $v \in V$  and the solution cost  $\text{cost}(\pi) = \sum_{v \in V} \text{cost}(v, \pi)/2$ .

In the tree for a vertex  $v$ , each node  $x$  keeps a value  $\gamma(x)$  (represented in the right bottom cell of each node in Figure 1). 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 the path between nodes  $y$  and  $z$  as  $\gamma(P(y, z)) = \sum_{x \in P(y, z)} \gamma(x)$ .

Let  $c_v^{\text{rev}}(l)$  be the cost incurred by inserting a vertex  $v$  into the position corresponding to a leaf  $l$  of the tree for  $v$ ; i.e., the sum of the costs of reverse edges connected with  $v$  when the

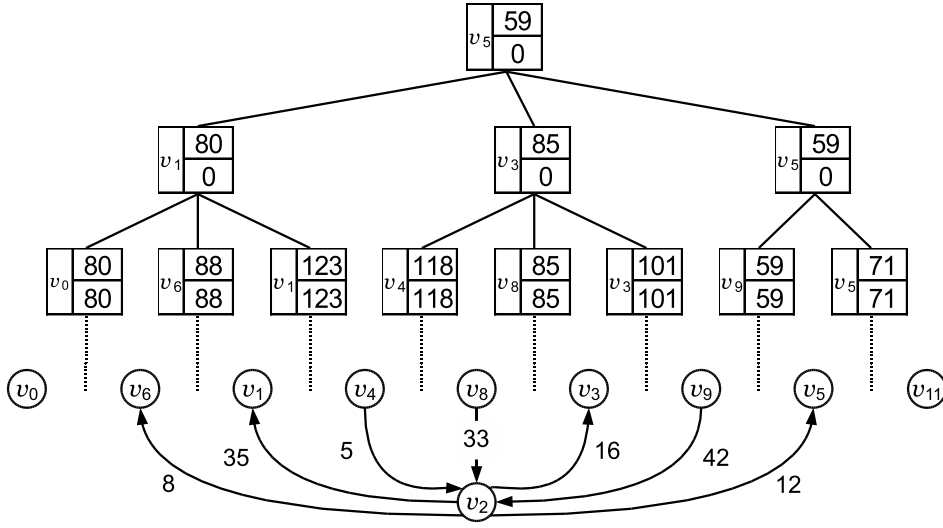


Figure 1: The vertices adjacent to vertex  $v_2$  and the tree corresponding to  $v_2$

position of  $v$  is in the gap defined by  $l$ . We control  $\gamma(x)$  so that  $c_v^{\text{rev}}(l) = \gamma(P(r_v, l))$  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  $\pi$  by inserting a vertex  $v$  into the position corresponding to a leaf  $l$ , we have  $\text{cost}(\pi'(v, l)) - \text{cost}(\pi) = c_v^{\text{rev}}(l) - \text{cost}(v, \pi)$ .

Two other values are kept in each node  $x$  of the tree. The first one,  $v_{\text{name}}(x)$ , carries the name of the vertex on the left of the gap represented by  $x$  if  $x$  is a leaf, or the value of  $v_{\text{name}}(y)$  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 1,  $v_{\text{name}}(x)$  is represented in the left cell of each node. By keeping the values of  $v_{\text{name}}(x)$  this way and using  $\pi^{-1}$ , we can find any leaf  $l$  in the tree of  $v$  by its  $v_{\text{name}}(l)$  in  $O(\log d_v)$  time.

The second one,  $\gamma_{\min}(x)$ , 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 1,  $\gamma_{\min}(x)$  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  $c_v^{\text{rev}}(l)$  among all the leaves in the tree of  $v$ , and we can find a solution with a smaller cost than the current one just by comparing the values of  $\gamma_{\min}(r_v)$  and  $\text{cost}(v, \pi)$  for all  $v \in V$ . If none of the trees for  $v$  have  $\gamma_{\min}(r_v)$  smaller than  $\text{cost}(v, \pi)$ , we can conclude that the current solution  $\pi$  is locally optimal.

The number of leaves in the tree of each vertex  $v$  is equal to  $d_v + 1$ , and 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  $\pi_{\text{init}}$ , 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  $\pi_{\text{init}}$ . 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 (2008).

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  $\gamma(l) = \gamma_{\min}(l) = \sum_{u \in V} c_{uv}$ . Then, we scan the list

of  $v$ , creating a leaf  $l$  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  $O(m \log \Delta)$ .

The list with the costs  $\text{cost}(v, \pi_{\text{init}})$  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  $\pi_{\text{init}}^{-1}$ .

## 2.2 Search and update of the data structure

To conduct a search through the neighborhood of a solution  $\pi$ , 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  $\text{cost}(v, \pi)$  that are kept in a list. This procedure can be done in  $O(n)$  time.

Once we find a  $v$  that satisfies  $\gamma_{\min}(r_v) < \text{cost}(v, \pi)$ , 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  $l$ , we look for the path with  $\text{cost}(P(r_v, l)) = \gamma_{\min}(r_v)$  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 of  $y$ ) until  $x$  is replaced by a leaf  $l$ .

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

After inserting  $v$  immediately after  $u$ , we have to update the trees for all the vertices  $w \in N(v)$ . Suppose  $\pi^{-1}(v) < \pi^{-1}(u)$  (the opposite case is similar). In the tree of 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  $\pi$  if  $u \notin N(w)$ . Then we update the values of  $c_v^{\text{rev}}(l)$  for all leaves  $l$  between  $l_v$  and  $l_u$ . The update for the tree of each  $w$  is executed through the following steps:

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  $\pi^{-1}(v_{\text{name}}(l_u)) \leq \pi^{-1}(u)$ . 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  $v_{\text{name}}(y) := v$  and  $\gamma(y) := \gamma_{\min}(y) := \gamma_{\min}(l_u)$ . (Inner nodes may be created according to the insertion operation for 2-3 trees.)
4. While  $x$  has a right sibling different from  $y$ , set  $x$  to its right sibling and then add  $c_{vw} - c_{wv}$  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  $x$ , set  $y$  to its left sibling and then add  $c_{vw} - c_{wv}$  to  $\gamma(y)$  and to  $\gamma_{\min}(y)$ . Update  $\gamma_{\min}(p(y))$ .
6. Set  $x := p(x)$  and  $y := p(y)$ . If  $x \neq y$ , repeat Steps 4 to 6; otherwise, update the values of 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  $O(\Delta \log \Delta)$  time to update all the necessary trees.

Let  $B(v, u)$  be the set of vertices in  $N(v)$  whose positions in  $\pi$  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  $\text{cost}(\pi)$  and  $\text{cost}(w, \pi)$  for all vertices  $w \in B(v, u)$ . The values of  $\text{cost}(\pi)$  and  $\text{cost}(v, \pi)$  are updated by subtracting  $\sum_{w \in B(v, u)} (c_{vw} - c_{vw})$  from both of them. To update the costs of the other vertices  $w$ , we subtract  $c_{vw} - c_{vw}$  from  $\text{cost}(w, \pi)$  for each  $w \in B(v, u) \setminus \{v\}$ . This cost update can be done in  $O(d_v)$  time.

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

Figure 2 shows the updates on the tree of  $v_2$  represented in Figure 1, with the new solution  $\pi'$  obtained by inserting  $v_1$  after  $v_9$ . In this case,  $c_{v_1 v_2} - c_{v_2 v_1} = -35$  and  $\text{cost}(v_2, \pi') = 50$ .

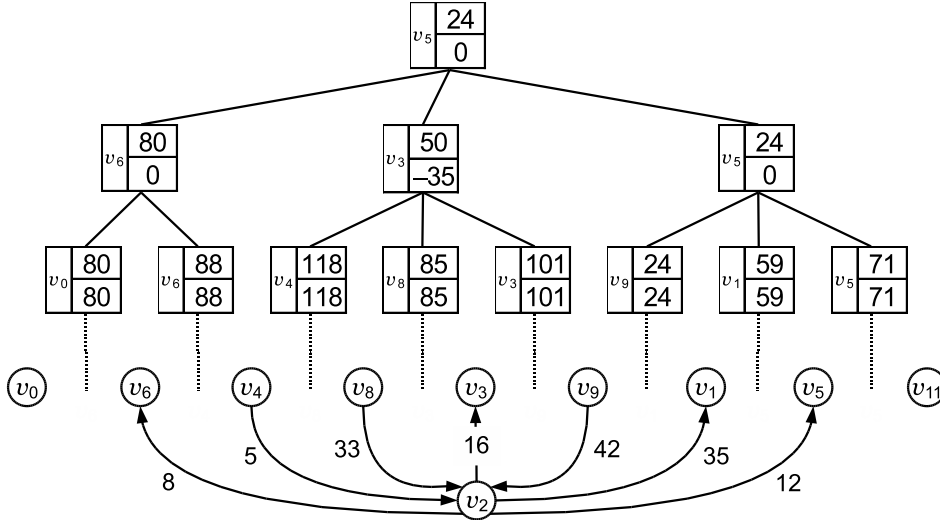


Figure 2: Updated tree for  $v_2$

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  $O(m \log \Delta)$  time.*

### 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<sup>1</sup>.

We compare the one-round time of TREE algorithm with the ones obtained by the methods proposed in Schiavinotto and Stützle (2004) and Sakuraba and Yagiura (2008), 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.

<sup>1</sup><http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html>



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 \(2008\)](#), 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

density $n$	1%		5%		10%		50%		100%	
	LIST	TREE	LIST	TREE	LIST	TREE	SCST	TREE	SCST	TREE
500	.000060	.0000089	.00086	.000025	0.0017	.000065	0.0027	.00069	0.0028	.0015
1000	.000235	.0000172	.00116	.000095	0.0082	.000246	0.0348	.00189	0.0347	.0038
2000	.001045	.0000465	.02428	.000282	0.0561	.000659	0.1822	.00457	0.1806	.0087
3000	.007508	.0000837	.06365	.000499	0.1351	.001130	0.4288	.00737	0.4271	.0143
4000	.019748	.0001261	.12070	.000723	0.2553	.001666	0.8898	.01042	0.8872	.0200
8000	.096189	.0003397	.51111	.001824	1.0851	.004011	4.7516	.02413	4.8053	.0466

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 valuable comments. 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

- Chanas, S. and Kobylański, P.: 1996, A new heuristic algorithm solving the linear ordering problem, *Computational Optimization and Applications* **6**, 191–205.
- Garcia, C. G., Pérez-Brito, D., Campos, V. and Martí, R.: 2006, Variable neighborhood search for the linear ordering problem, *Computers and Operations Research* **33**, 3549–3565.
- Grötschel, M., Jünger, M. and Reinelt, G.: 1984, A cutting plane algorithm for the linear ordering problem, *Operations Research* **32**, 1195–1220.
- Huang, G. and Lim, A.: 2003, Designing a hybrid genetic algorithm for the linear ordering problem, *Genetic and Evolutionary Computation - GECCO 2003, proc.* pp. 1053–1064.
- Sakuraba, C. S. and Yagiura, M.: 2008, A local search algorithm efficient for sparse instances of the linear ordering problem, *Japan-Korea Joint Workshop on Algorithms and Computation - WAAC 2008, proc.* pp. 44–50.
- Schiavinotto, T. and Stützle, T.: 2004, The linear ordering problem: Instances, search space analysis and algorithms, *Journal of Mathematical Modelling and Algorithms* **3**, 367–402.