A Memetic Algorithm for the Linear Ordering Problem with Cumulative Costs

Tao Ye\textsuperscript{a}, Kan Zhou\textsuperscript{a}, Zhipeng Lü\textsuperscript{a,*}, Jin-Kao Hao\textsuperscript{b}

\textsuperscript{a}School of Computer Science and Technology, Huazhong University of Science and Technology, 430074 Wuhan, P.R.China
\textsuperscript{b}LERIA, Faculty of Sciences, University of Angers, 2, Boulevard Lavoisier, 49045 Angers, France

Abstract

This paper introduces an effective memetic algorithm for the linear ordering problem with cumulative costs. The proposed algorithm combines an order-based recombination operator with an improved forward-backward local search procedure and employs a solution quality based replacement criterion for pool updating. Extensive experiments on 118 well-known benchmark instances show that the proposed algorithm achieves competitive results by identifying 46 new upper bounds. Furthermore, some critical ingredients of our algorithm are analyzed to understand the source of its performance.

Keywords: Linear Ordering; Memetic Algorithm; Local Search; Recombination Operator.

*Corresponding author.

Email addresses: yeetao@gmail.com (Tao Ye), zhipeng.lui@gmail.com (Zhipeng Lü), hao@info.univ-angers.fr (Jin-Kao Hao)
1. Introduction

Given a complete directed graph $G = (V, E)$ with nonnegative vertex weight $d_i$ and nonnegative arcs cost $C_{ij}$, where $V$ is the set of vertices ($n = |V|$), the Linear Ordering Problem with Cumulative Costs (LOPCC) aims to find a permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ of the $n$ vertices of $G$ such that the following function is minimized:

$$f(\pi) = \sum_{i=1}^{n} \alpha_{\pi_i}$$

where

$$\alpha_{\pi_i} = d_{\pi_i} + \sum_{j=i+1}^{n} C_{\pi_i \pi_j} \alpha_{\pi_j} \quad \text{for} \quad i = n, n-1, \ldots, 1$$

LOPCC was originally introduced in [1] to formulate a practical problem appeared in wireless communication systems. Since then, a number of solution approaches have been proposed to solve this problem. Benvenuto et al. proposed a heuristic approach based on greedy construction [2], which randomly selects a vertex from a reduced candidate list of the best available vertices and inserts the vertex into the permutation under construction. This algorithm is very fast, but its solution quality is generally unsatisfactory. Righini proposed an exact algorithm using branch-and-bound and a truncated branch-and-bound heuristic algorithm (TB&B) [3]. The truncation technique can reduce the running time by a factor of 10. Despite of its promising results, the high computational time cost of TB&B still prevented this algorithm from solving problems of size larger than $n = 35$ in a reasonable time.

On the other hand, a number of metaheuristic algorithms were proposed to solve the LOPCC problem and proved to be effective to find high-
quality solutions in a reasonable time. Some representative metaheuristic approaches include Iterated Local Search [4, 5], Tabu Search [6], Iterated Greedy-Strategic Oscillation and Path-Relinking [7]. Nevertheless, large instances (e.g., graphs with \( n \geq 100 \)) still represent a real challenge for all existing LOPCC approaches.

This paper presents for the first time a memetic algorithm for solving the LOPCC problem. The proposed algorithm integrates a new local search procedure within the framework of evolutionary computing. In particular, the proposed algorithm employs an order-based recombination operator and a solution quality based replacement strategy for population updating.

The performance of the proposed algorithm is assessed on a total of 118 LOPCC benchmark instances from the literature. Computational experiments show that the memetic algorithm matches the previous best solutions for 61 instances and improves the previous best solutions in 46 cases. To provide insights about the performance of the proposed algorithm, we analyze the influence of some critical ingredients of the algorithm.

The rest of this paper is organized as follows. Section 2 describes the key components of the memetic algorithm: The local search procedure, the recombination operator and the pool updating strategy. Section 3 gives the computational results and the comparison between the memetic algorithm and some state-of-the-art algorithms in the literature. Section 4 investigates several essential components of the proposed algorithm and conclusion is given in Section 5.
2. Memetic algorithm

2.1. Main Scheme

Memetic algorithms are known to be a powerful framework to solve hard combinatorial optimization problems (see e.g. [8, 9]). By combining the evolutionary framework with the local search procedure, memetic algorithms are expected to offer a balance between an intensified examination of a given search region and an exploratory discovery of new promising areas.

Following the general design principle of memetic algorithms [9], our memetic approach alternates between a recombination phase to generate new solutions and a local optimization phase to search around the newly generated solutions. Specifically, starting with a population of initial solutions, the algorithm repeats a number of evolution cycles (also called generations). At each generation, two parent solutions are randomly chosen from the current population. Then the recombination operator is applied to the parent solutions to generate an offspring solution which is subsequently optimized by the local search procedure. Finally, the population is updated with the improved offspring solution according to the solution quality of the offspring. This process is repeated until a stop condition (e.g., number of generations, computing time) is met.

The main scheme of the memetic algorithm is described in Algorithm 1, and the detailed descriptions of the four main components (i.e., population initialization, local search procedure, recombination operator and population updating strategy) are provided in the following subsections.
Algorithm 1 The Pseudo-code of the memetic algorithm

1: **Input**: The graph $G$
2: **Output**: The best solution found so far
3: $P = \{x^1, x^2, \ldots, x^p\} \leftarrow \text{randomly generate } p \text{ initial solutions} /\ast \text{Section 2.2} /$

4: for $i = 1, 2, \ldots, p$ do
5: \hspace{1em} $x^i \leftarrow \text{Local Search}(x^i) /\ast \text{Section 2.3} /$
6: end for
7: repeat
8: \hspace{1em} Randomly choose two individuals $x^a$ and $x^b$ from $P$
9: \hspace{1em} $x^0 \leftarrow \text{Recombination}(x^a, x^b) /\ast \text{Section 2.4} /$
10: \hspace{1em} $x^0 \leftarrow \text{Local Search}(x^0) /\ast \text{Section 2.3} /$
11: \hspace{1em} $P \leftarrow \text{Pool Updating} (x^0, P) /\ast \text{Section 2.5} /$
12: until stop condition is met
13: return the best solution found so far

2.2. Initial Population

The initial population is generated as follows. A random permutation is first created and then improved by the local search procedure (see Section 2.3). If the improved solution is not already present in the population, it is added into the population. Otherwise, this solution is discarded and a new random permutation is created. This procedure is iterated until the population is filled with $p$ solutions ($|P| = p$ is the population size).

2.3. Local Search Procedure

Given a solution $x$ (i.e., a permutation), we generate a neighboring solution by applying to $x$ an operator called insert, which moves a vertex from
its current position $i$ to another position $j$, denoted by \textit{insert}(i, j). This operator is widely used in the classical linear ordering problem (LOP) (see e.g. [10, 11]).

Our local search procedure is inspired by \textit{forward} local search in [7], and is divided into two parts: \textit{forward} and \textit{backward}. Instead of employing the traditional method (select the best solution from the whole neighborhood, which is very time-consuming), the local search procedure considers the vertex (denoted by $v^*$) with the maximum $\alpha$ value. We assume that the position of $v^*$ is $pos_{v^*}$. The set of target positions ($T_{pos}$) is composed of positions before $pos_{v^*}$. By employing \textit{insert} move, $v^*$ is moved from the current position to best target position chosen from $T_{pos}$ with respect to the objective function value. If there is no improving move associated with $v^*$, we turn to the next element with the maximum $\alpha$ value. This process is called the \textit{forward} part of our local search procedure, which is used in [7].

It should be noted that this procedure is much faster than the traditional local search method. The \textit{forward} part is repeated until the current solution cannot be further improved. The \textit{backward} part, which is opposite with the \textit{forward} part, selects the vertex with the minimum $\alpha$ value. The set of target positions ($T_{pos}$) includes positions after the position of the selected vertex. The \textit{backward} part is also repeated until the current solution cannot be further improved. This \textit{backward} part stops when it cannot further improve the solution quality any more. Note that this \textit{backward} part is missed in previous papers but it has significant influence on both solution quality and computational efficiency according to our experiments (see Section 4). The details of local search procedure is described in Algorithm 2.
To accelerate the evaluation of neighboring solutions, we employ a fast incremental evaluation technique introduced for LOP in [12]. The main idea is to maintain a special data structure to record the move values for swapping adjacent vertices. Particularly, each *insert* move is decomposed into several *swap* moves, which sequentially exchanges the vertices in adjacent positions. Experiments show that this method can reduce the computational time of the local search procedure by about 65%.

2.4. Recombination Operator

Previous studies suggest that the recombination operator is a key ingredient that determines the performance of the memetic algorithm. In this paper, we adopt the order-based operator which has been proven to be very useful for the classical LOP problem [12]. Specifically, two solutions in the current population are randomly selected as parent solutions. To maintain the diversity of the population, we restrict that the *similarity* (see Section 4) between the parent solutions should not be smaller than the average *similarity* of the solutions in the population.

Then, the order-based recombination operates in two phases. First, we copy one of the parents to the offspring solution. Second, we randomly select $k$ positions of the offspring solution and reorder the vertices on these $k$ positions according to their orders in another parent. Here, we set experimentally $k = n/2$ ($n$ being the number of vertices of the problem instance).

For example, we assume that $x^a = (2, 3, 1, 4, 6, 5)$ and $x^b = (4, 1, 2, 5, 6, 3)$ are the two parent solutions and 2,4,6 are respectively the selected positions. Then, the offspring solution obtained by the order-based recombination operator is $x^0 = (2, 4, 1, 5, 6, 3)$. 
2.5. Population Updating

The population updating strategy we employed is widely used in the literature (denoted by PoolWorst). This updating technique replaces the worst solution in the population with the new offspring solution if the offspring solution is good enough in terms of the objective value. Compared with some other updating strategies which consider both the solution quality and the diversity of the population, e.g., [13, 14, 15], PoolWorst strategy is able to accelerate the convergence speed, enabling the algorithm to find good solutions in relatively short time. Since our local search procedure is able to reach a balance between the search efficiency and effectiveness, it is more suitable for the updating strategy to accelerate the convergence speed.

3. Computational Results and Comparison

In this section, we assess the performance of the memetic algorithm on two sets of benchmark instances and compare it with the state-of-the-art algorithms in the literature.

3.1. Problem Instances and Experimental Protocol

Two sets of instances are used in the experiments. The first set of instances called LOLIB consists of 43 small instances with $n = 44$ to $60$. This set of instances is well known for the LOPCC problem. The second set of instances named RANDOM includes 75 instances. Specifically, 25 of them are small instances of size $n = 35$, and the other two parts are large and difficult instances of size $n = 100$ and $n = 150$. These instances are widely-used in a number of studies, see for example [6, 7].
The proposed algorithm is programmed in C++ and compiled using Microsoft Visual Studio 6.0 on a PC running Windows XP with 2.4 GHz CPU and 2 Gb RAM. The population size is set to 15 for all the tested instances. The stop condition for the algorithm is a fixed number of generations. For the instances of size $n = 150$, the generation limit is set to 200. For all the other instances, the generation limit is set to 100. Given the stochastic nature of the memetic algorithm, each instance is solved independently 10 times using different random seeds (some large instances are solved 5 times).

3.2. Computational Results

Tables 1 to 4 summarize the computational statistics of the memetic algorithm. In each table, column 1 gives the instance name. Columns 2 and 3 respectively give the number of vertices ($n$) and the previous best-known objective values ($f_{prev}$). Columns 4 – 6 report the statistics of the memetic algorithm: the best found objective value ($f_{best}$), the gap between our best results and the previous best-known results ($g_{best}$), the number of generations needed to reach the given $f_{best}$ ($iter$). If there are multiple runs reaching the best objective value, $iter$ stands for the average number of needed generations. Particularly, the number “0” means that the best found result is obtained during the initialization process. Column ($time_{best}$) gives the average computational time in seconds to detect the best objective values. The last column ($time_{total}$) shows the average total execution time in seconds. The last row named Average gives the average value for each column.
Table 1: Computational results on the 43 LOLIB instances

| Instance    | n  | $f_{prev}$ | Memetic Algorithm | $f_{best}$ | $g_{best}$ | iter | $time_{best}$ | $time_{total}$ |
|-------------|----|------------|-------------------|------------|-----------|------|---------------|---------------|
| be75ec      | 50 | 5.085      | 5.085             | 0          | 100       |      | 8.53          | 12.83         |
| be75np      | 50 | 16543433.910 | 16543433.910     | 0          | 91        | 146  | 10.16         | 11.77         |
| be75oi      | 50 | 2.788      | 2.788             | 0          | 146       |      | 10.16         | 11.77         |
| be75tot     | 50 | 297138.237 | 297139.434       | 1.197      | 29        | 137  | 23.84         | 27.35         |
| stabu1      | 60 | 13.284     | 13.284            | 0          | 137       |      | 23.84         | 27.35         |
| stabu2      | 60 | 14.029     | 14.029            | 0          | 89        | 20.03| 29.76         |               |
| stabu3      | 60 | 9.412      | 9.412             | 0          | 167       |      | 26.35         | 32.38         |
| t59b11xx    | 44 | 76261.813  | 76261.815         | 0.002      | 38        | 3.02 | 6.93          |               |
| t59d11xx    | 44 | 4086.303   | 4086.303          | 0          | 125       |      | 4.76          | 6.79          |
| t59e11xx    | 44 | 1618.897   | 1618.907          | 0.010      | 90        |      | 5.12          | 8.23          |
| t65b11xx    | 44 | 28230.444  | 28230.444         | 0          | 37        |      | 4.04          | 8.35          |
| t65d11xx    | 44 | 3898.568   | 3898.568          | 0          | 53        |      | 5.26          | 9.06          |
| t65i11xx    | 44 | 826108.100 | 826108.100       | 0          | 55        |      | 3.91          | 8.75          |
| t65i11xx    | 44 | 2657.735   | 2657.735          | 0          | 13        |      | 2.34          | 7.95          |
| t65l11xx    | 44 | 19.258     | 19.258            | 0          | 52        |      | 4.76          | 8.56          |
| t69r11xx    | 44 | 14.036     | 14.037            | 0.001      | 8         |      | 1.83          | 7.66          |
| t70b11xx    | 44 | 93.671     | 93.671            | 0          | 0.76      |      | 8.50          |               |
| t70d11xx    | 44 | 8.93       | 8.93              | 0          | 6         |      | 8.93          |               |
| t70f11xx    | 44 | 4.435      | 4.435             | 0          | 17        |      | 2.04          | 6.75          |
| t70k11xx    | 44 | 1.267      | 1.267             | 0          | 69        |      | 3.03          | 5.94          |
| t70l11xx    | 44 | 121146.029 | 121146.576       | 0.547      | 8         |      | 1.51          | 7.67          |
| t70m11xx    | 44 | 0.492      | 0.492             | 0          | 55        |      | 3.56          | 7.32          |
| t70n11xx    | 44 | 798.919    | 798.923           | 0.004      | 6         |      | 1.17          | 6.02          |
| t70p11xx    | 44 | 35490330260.185 | 35490477444.038 | 147183.852 | 16 | 1.63 | 6.63 |
| tiw56n54    | 56 | 2.645      | 2.645             | 0          | 1         |      | 3.42          | 19.85         |
| tiw56n58    | 56 | 3.620      | 3.620             | 0          | 85        |      | 12.43         | 17.1          |
| tiw56n62    | 56 | 3.024      | 3.024             | 0          | 79        |      | 10.63         | 20.07         |
| tiw56n66    | 56 | 2.687      | 2.687             | 0          | 5         |      | 3.27          | 16.87         |
| tiw56n72    | 56 | 1.877      | 1.877             | 0          | 45        |      | 7.54          | 15.43         |
| tiw56n72    | 56 | 1.567      | 1.567             | 0          | 47        |      | 13.12         | 21.51         |
| Average     |      | 48.7       | 825773080.516     | 825776503.437 | 79.603 | 36 | 6.92 | 12.58 |
Table 2: Computational results on the 25 RANDOM instances with $n = 35$

| Instance     | $n$ | $f_{\text{prev}}$ | Memetic Algorithm |          |         |         |
|--------------|----|-------------------|--------------------|----------|----------|----------|
| t1d35.1      | 35 | 0.923             | 0.923              | 0        | 44       | 1.33     | 3.14     |
| t1d35.2      | 35 | 0.167             | 0.167              | 0        | 27       | 1.76     | 1.96     |
| t1d35.3      | 35 | 0.154             | 0.154              | 0        | 7        | 0.54     | 2.01     |
| t1d35.4      | 35 | 0.196             | 0.196              | 0        | 41       | 1.06     | 2.03     |
| t1d35.5      | 35 | 1.394             | 1.394              | 0        | 14       | 0.69     | 1.87     |
| t1d35.6      | 35 | 0.200             | 0.200              | 0        | 67       | 1.27     | 2.33     |
| t1d35.7      | 35 | 0.120             | 0.120              | 0        | 29       | 0.96     | 2.74     |
| t1d35.8      | 35 | 0.226             | 0.226              | 0        | 11       | 0.79     | 2.03     |
| t1d35.9      | 35 | 0.436             | 0.436              | 0        | 62       | 1.27     | 2.23     |
| t1d35.10     | 35 | 0.205             | 0.223              | 0.018    | 6        | 0.51     | 1.67     |
| t1d35.11     | 35 | 0.369             | 0.369              | 0        | 0        | 0.43     | 0.58     |
| t1d35.12     | 35 | 0.234             | 0.234              | 0        | 38       | 1.03     | 2.07     |
| t1d35.13     | 35 | 0.196             | 0.196              | 0        | 9        | 0.56     | 1.87     |
| t1d35.14     | 35 | 0.138             | 0.138              | 0        | 37       | 0.94     | 1.93     |
| t1d35.15     | 35 | 1.376             | 1.376              | 0        | 9        | 0.64     | 2.21     |
| t1d35.16     | 35 | 0.286             | 0.286              | 0        | 11       | 0.67     | 2.12     |
| t1d35.17     | 35 | 0.199             | 0.199              | 0        | 1        | 0.48     | 2.99     |
| t1d35.18     | 35 | 0.381             | 0.381              | 0        | 7        | 0.57     | 2.43     |
| t1d35.19     | 35 | 0.236             | 0.236              | 0        | 128      | 2.14     | 2.62     |
| t1d35.20     | 35 | 0.068             | 0.068              | 0        | 43       | 1.13     | 2.23     |
| t1d35.21     | 35 | 0.202             | 0.202              | 0        | 10       | 0.87     | 2.76     |
| t1d35.22     | 35 | 0.177             | 0.177              | 0        | 5        | 0.43     | 1.77     |
| t1d35.23     | 35 | 0.345             | 0.345              | 0        | 28       | 0.94     | 2.06     |
| t1d35.24     | 35 | 0.132             | 0.132              | 0        | 12       | 0.53     | 1.7      |
| t1d35.25     | 35 | 0.143             | 0.143              | 0        | 9        | 0.67     | 1.93     |
| Average      | 35 | 0.341             | 0.341              | 0        | 11       | 0.84     | 2.17     |
Table 3: Computational results on the 25 RANDOM instances with $n = 100$

| Instance | $n$ | $f_{prev}$ | $f_{best}$ | $g_{best}$ | iter | $t_{best}$ | $t_{total}$ |
|----------|-----|------------|------------|-----------|------|-----------|-----------|
| t1d100.1 | 100 | 253.988    | 246.279    | -7.709    | 146  | 789       | 997       |
| t1d100.2 | 100 | 288.372    | 284.924    | -3.448    | 196  | 990       | 1060      |
| t1d100.3 | 100 | 1307.432   | 1236.237   | -71.195   | 142  | 771       | 993       |
| t1d100.4 | 100 | 7539.979   | 6735.661   | -804.318  | 196  | 913       | 948       |
| t1d100.5 | 100 | 169.336    | 162.423    | -6.913    | 196  | 1106      | 1136      |
| t1d100.6 | 100 | 395.035    | 391.662    | -3.373    | 189  | 763       | 784       |
| t1d100.7 | 100 | 5936.281   | 5641.137   | -295.144  | 170  | 798       | 1055      |
| t1d100.8 | 100 | 2760.619   | 2750.802   | -9.817    | 90   | 479       | 763       |
| t1d100.9 | 100 | 62.942     | 62.775     | -0.167    | 177  | 805       | 917       |
| t1d100.10| 100 | 162.942    | 159.126    | -3.816    | 195  | 960       | 1076      |
| t1d100.11| 100 | 233.586    | 230.810    | -2.776    | 112  | 567       | 996       |
| t1d100.12| 100 | 236.696    | 231.176    | -5.520    | 178  | 954       | 965       |
| t1d100.13| 100 | 593.319    | 578.307    | -15.012   | 187  | 888       | 919       |
| t1d100.14| 100 | 249.162    | 247.313    | -1.849    | 132  | 747       | 947       |
| t1d100.15| 100 | 406.478    | 408.312    | 1.834     | 192  | 1081      | 1158      |
| t1d100.16| 100 | 707.413    | 707.413    | 0         | 86   | 492       | 983       |
| t1d100.17| 100 | 725.790    | 718.920    | -6.87     | 156  | 801       | 853       |
| t1d100.18| 100 | 622.942    | 621.940    | -1.002    | 110  | 653       | 842       |
| t1d100.19| 100 | 228.486    | 227.374    | -1.112    | 130  | 674       | 1021      |
| t1d100.20| 100 | 255.151    | 238.586    | -16.565   | 128  | 791       | 913       |
| t1d100.21| 100 | 228.590    | 221.462    | -7.128    | 192  | 1131      | 1163      |
| t1d100.22| 100 | 159.336    | 141.255    | -18.081   | 111  | 676       | 1068      |
| t1d100.23| 100 | 1658.168   | 1656.877   | -1.291    | 196  | 881       | 935       |
| t1d100.24| 100 | 469.658    | 468.863    | -0.795    | 180  | 875       | 1091      |
| t1d100.25| 100 | 644.782    | 637.523    | -7.259    | 89   | 533       | 950       |

Average $100$ $1051.859$ $1000.286$ $-51.573$ $155$ $268$ $327$

Tables 1 and 2 show the computational statistics of the memetic algorithm on the small sized instances. For the 43 LOLIB instances and 25 RANDOM instances with $n = 35$, the memetic algorithm can achieve, respectively, 35 and 24 optimal values within a short time.
## Table 4: Computational results on the 25 RANDOM instances with $n = 150$

| Instance | $n$ | $f_{\text{prev}}$ | Memetic Algorithm | $f_{\text{best}}$ | $g_{\text{best}}$ | iter | $\text{time}_{\text{best}}$ | $\text{time}_{\text{total}}$ |
|----------|----|------------------|-------------------|------------------|------------------|------|----------------|------------------|
| t1d150.1 | 150 | 8588.289         | 8293.108          | -295.181         | 102              | 1075 | 1577           |
| t1d150.2 | 150 | 184953.686       | 159339.130        | -25514.556       | 172              | 1598 | 1819           |
| t1d150.3 | 150 | 574943.633       | 548507.282        | -26436.351       | 184              | 1604 | 1803           |
| t1d150.4 | 150 | 75510.287        | 68125.331         | -7384.956        | 199              | 1779 | 1817           |
| t1d150.5 | 150 | 79069.363        | 75426.662         | -3642.701        | 165              | 1609 | 1920           |
| t1d150.6 | 150 | 46829.985        | 46013.112         | -816.873         | 200              | 1688 | 1885           |
| t1d150.7 | 150 | 161149.153       | 150146.763        | -11002.390       | 200              | 1673 | 1731           |
| t1d150.8 | 150 | 251940.422       | 247564.438        | -4375.984        | 175              | 1606 | 1718           |
| t1d150.9 | 150 | 364320.250       | 363221.346        | -1098.904        | 197              | 1627 | 1675           |
| t1d150.10 | 150 | 122217.421       | 107685.011        | -14532.410       | 161              | 1500 | 1753           |
| t1d150.11 | 150 | 13900.039        | 12360.337         | -1539.702        | 190              | 1544 | 1729           |
| t1d150.12 | 150 | 65717.265        | 60614.534         | -5102.731        | 196              | 1676 | 1886           |
| t1d150.13 | 150 | 109460.320       | 105265.302        | -4195.018        | 160              | 1403 | 1906           |
| t1d150.14 | 150 | 74854.867        | 70153.934         | -4700.933        | 188              | 1786 | 1979           |
| t1d150.15 | 150 | 352880.286       | 321468.489        | -31411.797       | 193              | 1721 | 1779           |
| t1d150.16 | 150 | 1695096.691      | 16915821.128      | -34375.563       | 116              | 1114 | 1428           |
| t1d150.17 | 150 | 77828.419        | 74903.919         | -2924.500        | 182              | 1552 | 1786           |
| t1d150.18 | 150 | 711286.599       | 654737.416        | -56549.183       | 199              | 1571 | 1658           |
| t1d150.19 | 150 | 67840.414        | 66614.402         | -1226.012        | 193              | 1590 | 1733           |
| t1d150.20 | 150 | 1886041.875      | 2074926.337       | 188884.462       | 200              | 1907 | 1911           |
| t1d150.21 | 150 | 41453.911        | 39248.997         | -2204.914        | 151              | 1356 | 1604           |
| t1d150.22 | 150 | 695751.688       | 671281.287        | -24470.401       | 139              | 1128 | 1774           |
| t1d150.23 | 150 | 22203891.826     | 21468279.568      | -735612.258      | 199              | 1563 | 1844           |
| t1d150.24 | 150 | 105162.367       | 101072.915        | -4089.452        | 194              | 1844 | 1888           |
| t1d150.25 | 150 | 462316.511       | 467978.731        | 3482.220         | 153              | 1484 | 1703           |
| **Average** | 150 | 1827520.220      | 1795074.779       | -32445.444       | 176              | 1560 | 1772           |
Tables 3 and 4 report the computational results on the challenging large instances. For these instances, the memetic algorithm is able to improve a number of the previous best known results. Specifically, for the RANDOM instances with $n = 100$, the memetic algorithm improves the previous best known results for 23 out of 25 instances (indicated in bold). For one of the remaining instance ($t1d100.16$), the algorithm can match the previous best known result. While for the instance ($t1d100.15$), our result is worse than the previous best known one. When it comes to RANDOM instances of size $n = 150$ (Table 4), the memetic algorithm improves the previous best known results for 23 out of 25 instances (indicated in bold). The results for the two remaining instances are worse than the previous best known ones.

In summary, the computational results establish the efficiency and effectiveness of the proposed memetic algorithm. For the 68 small size instances, LOLIB instances and RANDOM instances with $n = 35$, the algorithm successfully matches the best-known results on most of the instances in a short time. While for the 50 challenging large instances, RANDOM instances with $n = 100$ and $n = 150$, the algorithm is able to find better solutions on 46 instances within a reasonable time.

3.3. Comparisons with the State-of-the-art Algorithms

In this section, the proposed memetic algorithm is compared with two state-of-the-art algorithms in the literature, respectively named TS [6] and EvPR [7]. The experiments are done on all the 118 instances. We try our best to conduct the comparison on a time-equalized basis, so we set the generation limit (the stop condition) to be 110 for RANDOM instances with $n = 150$, and 200 for the rest instances, and run the proposed algorithm on each
Table 5: Summarized comparison of the proposed algorithm with two state-of-the-art algorithms on all the instances

| Instances                  | TS         | EvPR      | Memetic   |
|----------------------------|------------|-----------|-----------|
| 43 LOLIB instances         | 8.26E+08   | **1.35**  | 8.26E+08  |
| Avg.deviation              | 1.40%      | 0.00%     | 0.00%     |
| Num.of opt                 | 28         | 36        | 34        |
| Total CPU seconds          | 37.74      | 32.34     | 12.50     |
| 25 random instances with n = 35 | 0.34   | 0.34      | 0.34      |
| Avg.deviation              | 0.51%      | 0.45%     | 0.35%     |
| Num.of opt                 | 21         | 24        | 24        |
| Total CPU seconds          | 3.40       | 3.75      | 2.17      |
| 25 random instances with n = 100 | 1161.46 | 1058.78   | 1034.30   |
| Avg.deviation              | 16.1%      | 5.8%      | 3.4%      |
| Num.of best ≤ 2            | ≤ 2        | ≤ 2       | 8         |
| Total CPU seconds          | 406.79     | 351.38    | 325.45    |
| 25 random instances with n = 150 | 2.27E+06 | 1.85E+06  | 1.90E+06  |
| Avg.deviation              | 11.64%     | 6.03%     | 7.07%     |
| Num.of best ≤ 2            | ≤ 2        | ≤ 2       | 1         |
| Total CPU seconds          | 2074.31    | 1127.24   | 1123.17   |

instance only once. Nevertheless, given the difference of computing platforms used for the compared algorithms, it is very difficult to replicate the exact stop conditions as those used in the reference algorithms. As a consequence, the comparisons shown in this section are for indicative purposes only and should be interpreted with caution.

Table 5 presents the results of these comparisons. Column 1 gives the instance name. Column 2 presents the four comparative criteria: the average objective value (Obj.function), average deviation w.r.t. the best-known result
(Ave.deviation), the number of instances where the algorithm can reach the optimal or best-known solution (Num.of best), and the average elapsed time on each instances (Total CPU seconds). Columns 3-5 respectively present the results obtained by TS, EvPR and memetic algorithm.

As one can observe from Table 5 when comparing with TS algorithm, memetic algorithm yields better results in general. One each set of instances, the Obj.function, Ave.deviation and Num.of best of the memetic algorithm are better than the TS algorithm. The Total CPU seconds of the two algorithms on each set of instances are at the same level.

When comparing the memetic algorithm with the EvPR algorithm, one can observe that the performance of the two algorithms are roughly the same on the first two sets of small instances. For the 25 RANDOM instances with $n = 100$, memetic algorithm’s performance is better. Its Obj.function and Avg.deviation are smaller than EvPR. However, when it comes to the 25 RANDOM instances with $n = 150$, memetic algorithm’s performance is worse than EvPR. The reason maybe that, in order to conduct the comparison on a time-equalized basis, we set the generation limit of the memetic algorithm on these instances to be 110, and within this time period, the memetic algorithm still focuses on exploration and therefore the computational results are relatively poor. These results show that the memetic algorithm may has some complementary feature w.r.t. the EvPR algorithm. If in some applications, the allowed computation time is limit, one should choose EvPR algorithm. Otherwise, if one focuses on finding high-quality solution and allows to use relatively longer computation time, choosing memetic algorithm may be better.
4. Analysis of the Forward-Backward Local Search Strategy

Local search is a critical component of the memetic algorithm. Our local search procedure is inspired by the local search procedure proposed in [7]. However, the local search procedure in [7] allows only forward moves, our local search procedure allows both forward and backward moves. In order to investigate whether this difference makes a meaningful contribution, we conduct experiments on some representative instances.

We keep other ingredients unchanged, and run two memetic algorithms, one with forward local search and the other with forward-backward local search, on 5 random instances (t1d100.1 t1d100.2 t1d100.11 t1d100.12 t1d100.21 t1d100.22). To reduce the impact of randomness, each algorithm is run 20 independent times. The stopping criterion is the execution time which is fixed to be 400s. We observe two characteristics: the average solution quality in the population and the population diversity.

Given two solutions $x^i$ and $x^j$, we define the distance between $x^i$ and $x^j$ (denoted by $d_{i,j}$) as: $n$ (the number of vertexes) – the length of the longest common subsequence of $x^i$ and $x^j$. Then, the population diversity (denoted by $pdi$) is defined as:

$$pdi = \frac{\sum_{i=1}^{p-1} \sum_{j=i+1}^{p} (d_{i,j})}{p(p-1)/2} \tag{3}$$

With this definition, $pdi = 0$ implies that all the solutions are the same. With the value of $pdi$ becoming smaller, the population diversity decreases, implying that solutions in the population tend to become more similar.

Fig.1 shows how the average solution quality (left) and the diversity of the population (right) evolving with the execution time. As Fig. 1 shows,
the algorithm with forward-backward LS can usually find better solutions, and meanwhile its convergence speed is faster.

5. Conclusion

In this paper, we have presented a memetic algorithm for the linear ordering problem with cumulative costs. The proposed algorithm uses an order-based recombination operator for generating new solutions and an effective local search procedure for local optimization. The proposed algorithm was evaluated on 118 well-known benchmark instances and the results proved its effectiveness and efficiency. For the 43 LOLIB instances and 25 small random instances, the algorithm reaches 35 and 24 optimal values within a short computational time. In addition, it is able to improve the previous best known objective values for 46 out of 50 large instances.
Acknowledgments

We are grateful for comments by the referees for their comments that help us improve the paper. This work was partially supported by the National Natural Science Foundation of China (Grant No. 61100144) and the RaDaPop (2009-2013) and LigeRo projects (2009-2013) from Pays de la Loire Region, France.

References

[1] Bertacco L, Brunetta L, Fischetti M. The linear ordering problem with cumulative costs. Eur J Oper Res 2008; 189(3): 1345-1357.

[2] Benvenuto N., Carnevale N., Tomasin S.. Optimum power control and ordering in SIC receivers for uplink CDMA systems. IEEE International Conference on Communications 2005, vol. 4, pages 2333-2337, 2005.

[3] Righini G. A branch-and-bound algorithm for the linear ordering problem with cumulative costs. Eur J Oper Res 2008; 186(3): 965-971.

[4] Villanueva DT, Huacuia HJF, Duarte A, Rangel RP, Valadez JMC, Soberanes HJP. Improving iterated local search solution for the linear ordering problem with cumulative costs (LOPCC). In: Setchi R, Jordanov I, Howlett RJ, Jain LC, editors. Knowledge-Based and Intelligent Information and Engineering Systems, Springer; 2010, p. 183-192.

[5] Villanueva DT, Huacuja HJF., Rangel RP., Valadez JMC., Soberanes HJP, Barbosa JJG. Iterated local search solution for the linear ordering problem with cumulative costs (LOPCC). In: Castillo O, Kacprzyk J,
Pedrycz W, editors. Soft Computing for Intelligent Control and Mobile Robotics, Springer; 2011, p. 395-404.

[6] Duarte A, Laguna M, MartíR. Tabu search for the linear ordering problem with cumulative costs. Comput Optim Appl 2011; 48(3): 697-715.

[7] Duarte A, MartíR, álvarez A, ángel-Bello F. Metaheuristics for the linear ordering problem with cumulative costs. Eur J Oper Res 2012; 216(2): 270-277.

[8] Moscato P. Memetic algorithms: a short introduction. In: Corne D, Dorigo M, Glover F, editors. New Ideas in Optimization, Maidenhead, UK: McGraw-Hill Ltd; 1999, p. 219-234.

[9] Hao JK. Memetic algorithms in discrete optimization. In: Neri F, Cotta C, Moscato P, editors. Handbook of Memetic Algorithms, Springer; 2012, p.73-94.

[10] Campos V, Glover F, Laguna M, MartíR. An experimental evaluation of a scatter search for the linear ordering problem. J Global Optim 2001; 21(4): 397-414.

[11] Garcia CG, Pérez-Brito D, Campos V, Martí R. Variable neighborhood search for the linear ordering problem. Comput Oper Res 2006; 33(12): 3549-3565.

[12] Schiavinotto T, Stützle T. The linear ordering problem: instances, search space analysis and algorithms. J Math Model Algorithm 2004; 3(4): 367-402.
[13] Lü Z, Glover F, Hao JK. A hybrid metaheuristic approach to solving the ubqp problem. Eur J Oper Res 2010; 207(3): 1254-1262.

[14] Lü Z, Hao JK. A memetic algorithm for graph coloring. Eur J Oper Res 2010; 203(1): 241-250.

[15] Porumbel D C, Hao J K, Kuntz P. An evolutionary approach with diversity guarantee and well-informed grouping recombination for graph coloring. Comput Oper Res 2010; 37(10): 1822-1832.
Algorithm 2 The pseudo-code of the local search procedure

1: **Input**: An initial solution \( x^0 \)
2: **Output**: A locally optimal solution
3: improved ← true
4: repeat
5:    improved ← false
6:    for \( i = 1, 2, 3, \ldots, n \) do
7:       Figure out the vertex \( v^* \) with the \( i \)th large \( \alpha \)
8:       Let \( pos_{v^*} \) be the position of \( v^* \)
9:       for \( j = pos_{v^*} - 1, \ldots, 1 \) do
10:          Swap the vertex on \( j \) and \( j + 1 \)
11:          Calculate the objective function value \( f' \)
12:          if \( f' \leq f_{best} \) then
13:             \( pos_{best} \leftarrow j \) and improved ← true
14:             \( f_{best} \leftarrow f' \)
15:          end if
16:       end for
17:       Move vertex \( v^* \) to position \( pos_{best} \)
18:    end for
19: until improved = false
20: improved ← true
21: repeat
22:    improved ← false
23:    for \( i = 1, 2, 3, \ldots, n \) do
24:       Figure out the vertex \( v^* \) with the \( i \)th small \( \alpha \)
25:       Let \( pos_{v^*} \) be the position of \( v^* \)
26:       for \( j = pos_{v^*} + 1, \ldots, n \) do
27:          Swap the vertex on \( j \) and \( j - 1 \)
28:          Calculate the objective function value \( f' \)
29:          if \( f' \leq f_{best} \) then
30:             \( pos_{best} \leftarrow j \) and improved ← true
31:             \( f_{best} \leftarrow f' \)
32:          end if
33:       end for
34:       Move vertex \( v^* \) to position \( pos_{best} \)
35:    end for
36: until improved = false