Designing a Hybrid Genetic Algorithm for the Linear Ordering Problem

Gaofeng Huang[1] and Andrew Lim[2]

[1] Department of Computer Science, National University of Singapore
3 Science Drive 2, Singapore 117543
[2] Department of Industrial Engineering and Engineering Management
Hong Kong Univ of Sci and Tech, Clear Water Bay, Kowloon, Hong Kong

Abstract. The Linear Ordering Problem(LOP), which is a well-known $\mathcal{NP}$-hard problem, has numerous applications in various fields. Using this problem as an example, we illustrate a general procedure of designing a hybrid genetic algorithm, which includes the selection of crossover/mutation operators, accelerating the local search module and tuning the parameters. Experimental results show that our hybrid genetic algorithm outperforms all other existing exact and heuristic algorithms for this problem.

Keywords: Linear Ordering Problem, Genetic Algorithm, Hybridization.

1 Introduction

The Linear Ordering Problem(LOP) has numerous applications in economics, archaeology, scheduling, the social sciences, and aggregation of individual preferences[5,6,7,8]. Of all these applications, the most famous one may be "the Triangulation for Input-Output Matrices"[7], which measures the movement of goods from one "sector" to another in economics research.

Mathematically, LOP can be formulated as :

$$\begin{array}{lll} \textbf{Instance} & : & \text{a matrix } C = \{c_{ij}\}_{n\times n} \\ \textbf{Solution} & : & p = (p_1, p_2, ..., p_n), \text{ a permutation of } 1...n \\ \textbf{Objective} & : & \text{to maximize } C(p) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} c_{p_i,p_j} \end{array}$$

This problem is known to be $\mathcal{NP}$-hard[9]. Many exact and heuristic algorithms have been proposed to solve it.

Several exact methods have been devised based on the integer programming technique. Grotschel *et al.* first proposed a cutting plane algorithm in [6,7]; Mitchell and Borchers[8] improved the result by combining the cutting plane with the interior point algorithm. However, all of these exact algorithms are extremely time-consuming.

E. Cantú-Paz et al. (Eds.): GECCO 2003, LNCS 2723, pp. 1053–1064, 2003.

On the other hand, heuristic algorithms seem to be more practical in solving large instances of this problem[1,2,3,4,5]. The heuristic proposed by Chanas and Kobylanski[5] can usually provide acceptable solutions both in terms of time and quality. Laguna *et al.*[1] applied Tabu Search technique with path relinking strategy on this problem successfully, while Campos *et al.*[2] used Scatter Search to solve this problem. These two approaches are often regarded as the best at this moment.

The purpose of this paper includes: (1) to develop an effective heuristic for LOP and (2) to illustrate a general procedure of designing a hybrid genetic algorithm. Genetic algorithms(GA) have shown to be competitive technique for solving general combinatorial optimization problems. However, it is possible to incorporate problem-specific knowledge into GA so that the results can be further improved. The hybridization between GA and Local Search is such a method. Experimental results show that our hybrid genetic algorithm outperforms all other existing exact and heuristic algorithms for LOP.

The rest of this paper is organized as follows. We first present the details of our hybrid GA implementation in Section 2. In Section 3, preliminary experiments are conducted to tune our algorithm. The final computational results and comparisons are reported in Section 4. In the last section, we present our conclusions.

2 Hybrid Genetic Algorithm

Many variations of hybridization between Genetic Algorithms and Local Search have been proposed[11,13,14]. In this paper, our hybrid GA has the following structure:

Algorithm 1 Hybrid Genetic Algorithm

```
1: Initialize population_size, crossover_rate, mutation_rate
2: Generate Initial Generation gen
3: while not TerminateConditions() do
4:   for pop ← 1 to population_size do
5:     Randomly selected two individuals from gen according to the fitness, say, gen_x and gen_y
6:     if random[0,1) < crossover_rate then
7:       next_gen_pop ←CrossoverOperator(gen_x, gen_y)
8:     else
9:       next_gen_pop ←Copy(better(gen_x, gen_y))
10:    end if
11:    if random[0,1) < mutation_rate then
12:      next_gen_pop ←MutationOperator(next_gen_pop)
13:    end if
14:    next_gen_pop ←LocalSearch(next_gen_pop)
15:    if fitness(next_gen_pop) > history_best then
16:      history_best ←fitness(next_gen_pop)
17:      solution ← next_gen_pop
18:    end if
19:  end for
20:  gen ← next_gen
21: end while
```

In this structure, we include all the basic components of a typical genetic algorithm, such as Initial Generation, Terminate Conditions, CrossoverOperator, MutationOperator, Fitness Function etc. In addition, we have also incorporated a local search module to improve the quality of each individual in the population. More details are provided in the following subsections.

2.1 Genetic Algorithm

- **Individual Representation and Fitness Function :** In LOP, any permutation $p = (p_1 p_2 ... p_n)$ is a feasible solution. It is very natural to use a n length vector p as the representation (chromosome) of an individual. The objective function $C(p)$ acts as the fitness function of chromosome.
- **Initial Generation :** For the initial generation, each individual is set using a permutation that is randomly generated, after which a local search is applied to it to improve its quality. Therefore, every individual in the initial generation is already a local maximum in the solution space.
- **Terminate Conditions :** We do not have duplicate detection scheme in our genetic algorithm. The evolution process is to converge to a "best" solution. When this happens, the algorithm terminates. However, checking for convergence can be time consuming. An alternative is to use the fitness function to approximate convergence.
- **Crossover Operators :** Three crossover operators, including PMX, CX and OX, are implemented in our algorithm and tested by our experiments.
  - Partially Mapped Crossover(PMX) operator was proposed by Goldberg and Lingle[10]. The basic idea of PMX is to exchange a partial segment between two parents. However, a skill named mapping is used in order to keep the results still as feasible permutations.
  - Cycle Crossover(CX) operator was first used in [12]. It first finds all the mapping cycles between two parents. After that, for each cycle, it randomly selects one of the two parents, and copies its elements to the offspring in corresponding position.
  - Order Crossover(OX) operator[13] first randomly selects several same elements in both parents, and then exchanges are made between two parents in those positions in order.
- **Mutation Operators :** We experiment with the two most commonly used mutation operators, DM and k-EM.
  - Displacement Mutation(DM) operator[14] randomly select a segment from the chromosome sequence and insert it into another randomly selected position.
  - k-EM is a variation of Exchange Mutation(EM) operator, where k exchange(swap) operations are performed synchronously in $2 \times k$ random selected positions.

2.2 Local Search

The local search strategy we used in LOP is based on the idea of iterative improvement. It starts with an initial solution(permutation p), and tries to improve the solution via a series of neighbour moves until no improvement can be made.

Neighbouring Move : In each iteration, the solution p is improved by a "neighbouring move". We use the INSERT move (also named DELETE-INSERT or SHIFT) and the EXCHANGE move (also named SWAP or Pairwise-Interchange) that are commonly used in permutation problems. For more details please see Table 1.

Table 1. Two commonly used neighbouring moves

neighbouring move	explanation	examples: $p =$ (5, 3, 1, 2, 4)
INSERT(p, i, j)	insert p_i to position j	INSERT$(p, 4, 1) =$ (2, 5, 3, 1, 4)
EXCHANGE(p, i, j)	exchange p_i and p_j	EXCHANGE$(p, 4, 1) =$ (2, 3, 1, 5, 4)

The following is an interesting observation.

Theorem 1. *For the LOP,* INSERT *move subsumes* EXCHANGE *move.*

Proof. – **If a solution p can be improved by the INSERT move, it may not be improved by the EXCHANGE move.**
An easy counterexample can be constructed when $n = 3$:
consider $p = (1, 3, 2)$, $c_{2,1} - c_{1,2} = 10$, $c_{3,1} - c_{1,3} = -1$, $c_{2,3} - c_{3,2} = -100$,
An INSERT move can be made to improve the result since
INSERT$(p, 1, 3) - p = C(3, 2, 1) - C(1, 3, 2) = (c_{3,1} - c_{1,3}) + (c_{2,1} - c_{1,2}) = -1 + 10 = 9 > 0$.
Consider the EXCHANGE move,
EXCHANGE$(p, 1, 2) - p = C(3, 1, 2) - C(1, 3, 2) = -1 < 0$,
EXCHANGE$(p, 1, 3) - p = C(2, 3, 1) - C(1, 3, 2) = -100 + 10 - 1 < 0$,
EXCHANGE$(p, 2, 3) - p = C(1, 2, 3) - C(1, 3, 2) = -100 < 0$.
Hence, no EXCHANGE move can be made to improve the solution p.

– **If a solution p can be improved by the EXCHANGE move, it can always be improved by the INSERT moves.**
Suppose p can be improved by EXCHANGE$(p, i, j)(i < j)$:

$$\begin{aligned} \text{Let } p'' &= \text{INSERT}(p, i, j-1) \\ p''' &= \text{INSERT}(p, j, i) \\ \text{Obviously, } p' = \text{EXCHANGE}(p, i, j) &= \text{INSERT}(p'', j, i) = \text{INSERT}(p''', i+1, j) \\ \text{So, } C(p') - C(p) &= (C(p') - C(p'')) + (C(p'') - C(p)) \\ &= (C(p') - C(p''')) + (C(p''') - C(p)) \end{aligned}$$

A important property of LOP is that

$$C(p') - C(p''') = C(p'') - C(p) = \sum_{k=i+1}^{j-1} (c_{p_k, p_i} - c_{p_i, p_k})$$

Therefore, $C(p') - C(p) = (C(p'') - C(p)) + (C(p''') - C(p))$.
Since solution p' is better than p, $C(p') - C(p) > 0$, we have $C(p'') - C(p) > 0$ or $C(p''') - C(p) > 0$, that is, p can also be improved by INSERT. □

As a result, only INSERT moves are considered in our algorithm. The gain in objective function after one INSERT move is given as:

$$dC_{i,j} = C(\text{INSERT}(p,i,j)) - C(p) = \begin{cases} \sum_{k=i+1}^{j}(c_{p_k,p_i} - c_{p_i,p_k}) & \text{for } i < j \\ 0 & \text{for } i = j \\ \sum_{k=j}^{i-1}(c_{p_i,p_k} - c_{p_k,p_i}) & \text{for } j < i \end{cases} \quad (1)$$

Search Strategy : In each iteration, there are $n(n-1)$ choices for an INSERT move. The FirstFit and BestFit are two widely adopted strategies.

As shown in Algorithm 2, the FirstFit search strategy always uses the first found move which can lead to a better solution.

Algorithm 2 FirstFit

```
1: p ← Initialize Solution(permutation)
2: finishFlag ← false
3: while not finishFlag do
4:    finishFlag ← true
5:    for i ← 1 to n do
6:       for j ← 1 to n do
7:          compute dC_{i,j} using Equation(1)
8:          if dC_{i,j} > 0 then
9:             p ← INSERT(p, i, j) {update the solution}
10:            finishFlag ← false
11:         end if
12:      end for
13:   end for
14: end while
```

In this algorithm, the computation of $dC_{i,j}$ using Equation(1) is very time-consuming, since each entry takes O(n) time. If the "WHILE-LOOP" runs T_1 times, the whole algorithm will need $T_1 n^2 O(n) = T_1 O(n^3)$ time.

$$dC_{i,j} = \begin{cases} dC_{i,j-1} + (c_{p_j,p_i} - c_{p_i,p_j}) & \text{for } i < j \\ 0 & \text{for } i = j \\ dC_{i,j+1} + (c_{p_i,p_j} - c_{p_j,p_i}) & \text{for } j < i \end{cases} \quad (2)$$

Unlike FirstFit, BestFit search strategy uses Equation(2) to compute all n^2 $dC_{i,j}$ entries and finds the best solution among all n^2 possible moves. Since each entry can be computed in O(1) time, if the whole algorithm terminates after t_2 iterations, it will take $t_2 O(n^2)$ time.

It's expected that $t_2 < T_1 n^2$ because BestFit tends to take the "faster" ascent direction to a local maximum. On the other hand, usually $t_2 \gg T_1$ since in each "round", FirstFit has up to n^2 INSERT moves, while BestFit makes 1 best INSERT move. This may explain the results in [1] that BestFit even takes more real CPU time than FirstFit.

We propose a FastFit search strategy that takes advantage of the strengths of both approaches. By using the idea of the "cache", we set a "dirty" flag in our algorithm. As long as no actual INSERT move is made, the $dC_{i,j}$ cache

Algorithm 3 FastFit

```
1: p ← Initialize Solution(permutation)
2: finishFlag ← false
3: while not finishFlag do
4:    finishFlag ← true
5:    for i ← 1 to n do
6:       dirtyFlag ← false
7:       dC_{i,i} ← 0
8:       for j ← i+1, i+2, ..., n; i−1, i−2, ..., 1 do
9:          if dirtyFlag then
10:            compute dC_{i,j} using Equation(1)
11:            dirtyFlag ← false
12:         else
13:            compute dC_{i,j} using Equation(2)
14:         end if
15:         if dC_{i,j} > 0 then
16:            p ← INSERT(p, i, j) {update the solution}
17:            finishFlag ← false
18:            dirtyFlag ← true
19:         end if
20:      end for
21:   end for
22: end while
```

is always "clean", so we can compute the next dC entry in O(1) time. Only after one actual INSERT move, the dC cache becomes "dirty" so that O(n) time is needed to compute $dC_{i,j}$. If the "WHILE-LOOP" runs T_3 times, the whole FastFit algorithm would need $T_3O(n^2) + t_3O(n)$, where t_3 is the number of INSERT moves made to reach a local maximum.

It is expected that $T_3 \approx T_1$, as FastFit and FirstFit are very similar. We have $t_2 \gg T_1$, so $T_3 \ll t_2$. Therefore, FastFit is expected to be much faster than BestFit and FirstFit.

3 Preliminary Experiments

A typical Genetic Algorithm may have 100 individuals, while `crossover_rate` and `mutation_rate` are set to be 0.8 and 0.05 respectively. However, performance of a Genetic Algorithm may be very sensitive to the settings of these parameters. And there also may be some interaction effects between crossover type, mutation type, crossover rate and mutation rate. Since it's too computationally exhaustive to experiment all such combinations, therefore, in this section, preliminary experiments are conducted to tune our GA with FastFit local search strategy.

25 instances from [2] are used for our preliminary experiments. These instances are the same as those instances with $n = 75$ in "Random Instances Set B"(see Section 4.3).

3.1 Combination of Crossover Operators and Mutation Operators

First, we experiment the combination of crossover and mutation operators with the configuration : `crossover_rate` = 0.8, `mutation_rate` = 0.05,

Table 2. Combination of Crossover operators and Mutation operators

	OX	CX	PMX	average	std dev
DM	32916465.40	32915105.08	32916825.64	32916132.04	907.43
1-EM	32916657.56	32916144.60	32916548.52	32916450.23	270.24
2-EM	32916467.28	32915031.28	32917855.44	32916451.33	1412.15
5-EM	32917524.20	32915519.84	**32918195.64**	32917079.89	1392.13
10-EM	32917299.60	32914183.04	32917273.64	32916252.09	1791.90
average	32916882.81	32915196.77	32917339.78		
std dev	495.66	718.71	687.97		

`population_size` = 100. Table 2 shows the average results of all 25 instances for each combination.

From Table 2, the performance of different operators do not vary much. Among these, the combination of crossover operator PMX and mutation operator 5-EM provides the best average result. However, PMX operator seems to be slightly better than the other two crossover operators.

3.2 Tuning of Parameter `mutation_rate`

Using the combination of PMX and 5-EM operator(denoted as GA_PMX_5-EM), now we consider the `mutation_rate` parameter with `crossover_rate` = 0.80 and `population_size` = 100.

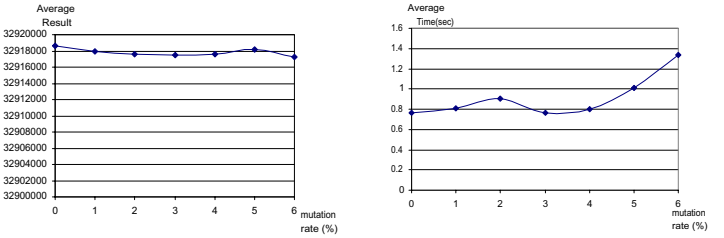

Fig. 1. Tuning of Parameter `mutation_rate`

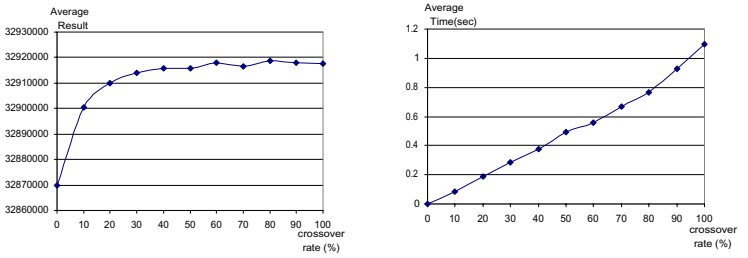

Fig. 2. Tuning of Parameter `crossover_rate`

Figure 1 shows the average results and CPU time taken for different `mutation_rate`. It's reasonable that CPU time increases with the growth of

mutation_rate, because more mutation means more computation. But as you can see in Fig. 1, the average result is nearly the same, which means the mutation in our GA does not improve the solution's quality significantly, hence we set mutation_rate = 0.

3.3 Tuning of Parameter crossover_rate

Parameter crossover_rate determines the probability of crossover happens in GA. Fig. 2 shows the average results and consumed CPU time under configuration : GA_PMX, mutation_rate = 0, population_size = 100.

As shown in Fig. 2, the average result increases with the growth of crossover_rate, especially when it is less than 0.5. On the other hand, the CPU time increases with crossover_rate in a approximately linear manner. With the compromise of time and solution quality, we choose crossover_rate = 0.5.

3.4 Tuning of Parameter population_size

population_size may be the most influential parameter in a Genetic Algorithm. If population_size is too small, there will be insufficient genetic diversity to search the feasible solution space thoroughly. On the other hand, the algorithm may be extremely slow when population_size is too large.

Fig. 3 shows the preliminary results with different population_size under the configuration : GA_PMX, no mutation, crossover_rate = 0.50. With the growth of population_size, the average result increase consistently although the increase becomes slower, while the CPU time consumed is approximately linear to population_size. Therefore, it's reasonable to choose population_size = 40, as it provide a good balance between solution quality and computing time.

3.5 Summary: Effect of Genetic Algorithm and Local Search

In our preliminary experiments, we examine the effects of both the Genetic Algorithm and Local Search. The following four algorithms are compared:

- SimpleGA is the simple Genetic Algorithm without local search.
- LocalSearch(random) is a multi-round local search algorithm. In each round, it starts with a random initial permutation and uses FastFit strategy to improve the solution.
- hGA(untuned) is a hybrid Genetic Algorithm with the configuration : GA with OX crossover operator, DM mutation operator, BestFit local search strategy, crossover_rate = 0.8, mutation_rate = 0.05, population_size = 100. This stands for an untuned hybrid Genetic Algorithm.
- hGA(tuned) is our final hybrid Genetic Algorithm with the configuration : GA with PMX crossover operator, no mutation operator, FirstFit local search strategy, crossover_rate = 0.5, population_size = 40.

Fig. 4 shows the time-quality curve for these 4 algorithms. The vertical axis shows the CPU time consumed, while the horizontal axis shows the average result for the 25 instances that we experimented with.

The following remarks can be made from Fig. 4:

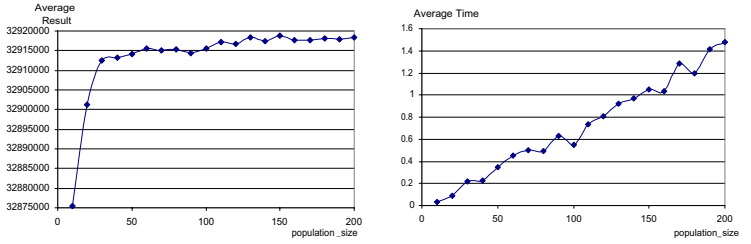

Fig. 3. Tuning of Parameter `population_size`

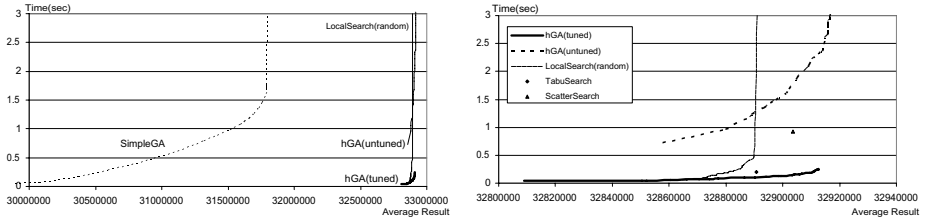

Fig. 4. Time-Quality Curve

- SimpleGA is rather bad in performance.
- LocalSearch(random) has the ability to find relatively good solutions in a short time. Unfortunately, even if much more time is given, this algorithm cannot improve the best solution by much. For the 25 instances that we have tested, no better solutions can be found after running 0.5 second.
- hGA(untuned) shows the power of hybridization of Genetic Algorithm and Local Search. Given enough time, it is capable of finding competitive solutions. However, it's slower than LocalSearch(random) since even the initialization alone will take more than 0.5 second.
- hGA(tuned) is about 10 times faster than hGA(untuned), which shows the power of proper tuning a search method. Consequently, hGA(tuned) outperforms Tabu Search ans Scatter Search both in time and solution quality.

Guided by the Genetic Algorithm, it is possible for Local Search to improve the quality of solution consistently; and with the help of Local Search, Genetic Algorithm becomes competitive in finding good solutions.

4 Computational Results

Three widely-used sets of instances are tested to demonstrate the effectiveness of our hybrid Genetic Algorithm. All the codes are implemented in C/C++ and run on a PentiumIII 800Mhz PC. And for each instance, our hGA is run only once.

However, the computing machines used in previous papers are different, i.e., Intel Pentium 166Mhz PC, Intel PentiumIII 500Mhz, Sun SPARC 20 Model 71. Therefore, in order to compare the CPU time, a scaling scheme is used according to SPEC(`http://www.specbench.org/osg/cpu2000/`)[1]

4.1 LOLIB Instances

LOLIB may be the most widely-used testing instances for LOP. It contains 49 instances of real-world input-output tables from sectors in the European and United States Economies. The data together with optimal solutions can be obtained from `www.iwr.uni-heidelberg.de/groups/comopt/software/LOLIB/`

Nearly all papers on LOP use LOLIB as test instances: in [1], the results of Chanas-Kobylanski algorithm(CK) and Tabu Search with path relinking(TS-LOP) are reported. [2] shows computational results of two Scatter Search versions(SS, SS10). A Variable Neighborhood Search(VNS-LOP) algorithm is applied in [3], while [4] gives the results of a Lagrangian Based Heuristic(LH-VP).

Table 3. LOLIB (49 instances)

	Avg.Obj. Value	Deviation from Opt.	No. of Optimal	Avg.CPU Times(sec) in P166	PIII500	PIII800
CK	22018008.35	0.15%	11	0.1		
SS	22041229.8	0.01%	42	3.82		
SS10	22041232.3	0.01%	43	14.28		
LH-VP		0.00%	43		5.581	
VNS-LOP	22041260.8	0.00%	44	0.87		
TS-LOP	22041261.51	0.00%	47	0.93		
hGA	22041263.82	—	49	< 0.2	< 0.05	0.0247

Table 3 shows for each algorithm the average objective function value over all 49 instances, the average percent deviation from the optimal solution, the number of optimal solutions a particular algorithm can reach, and the average CPU time. It's evident that our hGA outperforms all other algorithms, since not only is it faster, but also it's the only one algorithm which can provide all optimal solutions for 49 instances.

4.2 Random Instances Set A

This set of instances is randomly generated by J. E. Mitchell and B. Borchers[8]. There are 30 instances, where the size of instances vary from 100 to 250. More specifically, there are 5 instances with size $n = 100$ and $n = 250$ respectively, and 10 instances with size $n = 150$ and $n = 200$. Both the instances and optimal values are available at the website: `www.rpi.edu/~mitchj/generators/linord/`

The results of Interior Point algorithm, Simplex Cutting Plane algorithm and the combined of these two are reported in [8], while in [4] two Lagrangian Based Heuristics(LH-PC and LH-VP) are applied to this set of instances.

[1] SPEC(Standard Performance Evaluation Corporation) points out that PIII 800 is not more 2 times faster than PIII500, not more 8 times faster than P166 and not more 12 times faster than SPARC 20/71

Table 4. Random Instances Set A (30 instances)

	Avg.Obj. Value	Deviation from Opt.	No. of Optimal	Avg.CPU Times(sec) SPARC20	PIII500	PIII800
Interior	—		26	2390.11		
Simplex	—		30	2697.33		
Combined	—		30	356.16		
LH-PC		0.227%	16		605.13	
LH-VP		0.014%	28		105.46	
hGA	777324.17	—	30	< 24	< 4	1.9665

As you can see from Table 4, our hGA is $10 \sim 100$ times faster than those exact algorithms and more than 25 times faster than Lagrangian Based Heuristics. What's more important, with respect to all these 30 instances, the solutions obtained by our hGA are all optimal.

4.3 Random Instances Set B

This set includes 75 instances generated by Laguna *et al.*[1][2], which consists of 25 instances for each problem size $n = 75, 150, 200$ with each entries of the cost matrix c_{ij} randomly distributed in (0, 25000). No optimal solutions are reported for these instances by now, while the Tabu Search with path relinking strategy (TS-LOP) gives the best result before our experiment.

Table 5. Random Instances Set B (75 instances)

	Avg.Obj. Value	Deviation from best	No. of best	Avg. Times(sec) in P166	PIII800
CK	128663947.3	0.64%	0	10.67	
CK-10	128919838.0	0.39%	0	108.44	
TS-LOP	129269367.5	0.11%	5	20.19	
hGA	129437686.3	—	75	< 18	2.2402

Table 5 shows the results of our experiment. As you can see, our hGA algorithm gives better solutions than TS-LOP for 70 instances, for the other 5 instances the solutions between hGA and TS-LOP are same. It is clear that our hGA outperforms TS-LOP.

5 Conclusion

The Linear Order Problem is studied in this paper. We designed a hybrid Genetic Algorithm by integrating Genetic Algorithm and Local Search strategy successfully. Experiments indicate that parameters are influential for genetic algorithm. We described the procedure of tuning the parameters and develop a FastFit algorithm to accelerate the local search. As a result, the algorithm after tuning becomes 10 times faster than before.

Computational results show that our hybrid genetic algorithm outperforms all other existing exact and heuristic algorithms.

[2] The authors would like to express special thanks to Prof. Manuel Laguna for providing the instances and detail results of TS-LOP.

References

1. M. Laguna, R. Martí and V.Campos: *Intensification and diversification with elite tabu search solutions for the linear ordering problem*, Computer and Operation Research, vol.26, pp. 1217–1230, (1999)
2. V. Campos, M. Laguna and R. Martí: *Scatter Search for the Linear Ordering Problem*, New Ideas in Optimization, D. Corne, M. Dorigo and F. Glover (Eds.), McGraw-Hill, pp. 331–339 (1999)
3. Carlos García González and Dionisio Pérez-Brito: *A Variable Neighborhood Search for Solving the Linear Ordering Problem*, In the Proceedings of MIC'2001 - 4th Metaheuristics International Conference, pp. 181–185, Porto, Portugal, (2001)
4. Alexandre Belloni and Abilio Lucena: *Lagrangian Based Heuristics for the Linear Ordering Problem*, In the Proceedings of MIC'2001 - 4th Metaheuristics International Conference, pp. 445–449, Porto, Portugal, (2001)
5. S. Chanas and P. Kobylanski: *A New Heuristic Algorithm Solving the Linear Ordring Problem*, Computational Optimization and Applications, vol.6, pp. 191–205, (1996)
6. M. Grotschel, M. Junger and G. Reinelt: *A Cutting Plane Algorithm for the Linear Ordering Problem*, Operations Research, vol.32, no.6, pp. 1995–1220, (1984)
7. M. Groetschel, M. Juenger and G. Reinelt: *Optimal Triangulation of Large Real World Input-Output Matrices*, Statistische Hefte 25, 261–295, (1984)
8. J. E. Mitchell and B. Borchers: *Solving linear ordering problems with a combined interior point/simplex cutting plane algorithm*, High Performance Optimization, Kluwer Academic Publishers, Dordrecht, The Netherlands, H. Frenk et al. (Eds.), pp. 349–366, (2000)
9. R.M. Karp: *Reducibility among combinatorial problems*, In R.E. Miller and J.W. Thatcher (Eds.), Complexcity of Computer Computations, pp. 85–103, New York, (1972)
10. D. Goldberg and R. Lingle: *Alleles, loci, and the traveling salesman problem*, in the Proceedings of the 1st International Conference on Genetic ALgorithms and Their Applications, pp. 154–159, Lawrence Erlbaum, Hillsdale, New Jersey, (1985).
11. D. Goldberg: *Genetic Algorithms in Search, Optimization and Machine Learing*, Addison-Wesley, Reading, MA, (1989)
12. I. Oliver, D. Smith, and J. Holland: *A study of permutation crossover operators on the tsp*, in the Proceedings of the 2nd International Conference on Genetic ALgorithms and Their Applications, pp. 224–230, Hillsdale, New Jersey, (1987).
13. G. Syswerda: *Schedule Optimization Using Genetic Algorithms*, in L.Davis (Ed.) Handbook of Genetic ALgorithms, pp. 332–349, New York, (1991)
14. Z. Michalewicz: *Genetic Algorithms + Data Structure = Evolution Programs*, Springer-Verlag, Berlin Heidelberg, (1992)