A competitive genetic algorithm for single row facility layout

Ravi Kothari*, Diptesh Ghosh

P&QM Area, IIM Ahmedabad, Vastrapur, Ahmedabad 380015, Gujarat, INDIA

Abstract

The single row facility layout is the NP-Hard problem of arranging facilities with given lengths on a line, so as to minimize the weighted sum of the distances between all pairs of facilities. Owing to the computational complexity of the problem, researchers have developed several heuristics to obtain good quality solutions. In this paper, we present a genetic algorithm to solve large SRFLP instances. Our computational experiments show that an appropriate selection of genetic operators can yield high quality solutions in spite of starting with an initial population that is largely randomly generated. Our algorithm improves the previously best known solutions for the 24 instances of 43 benchmark instances and is competitive for the remaining ones.

Keywords: Facilities planning and design; Single Row Facility Layout; Genetic algorithm

1. Introduction

The single row facility layout problem (SRFLP) is the problem of obtaining a linear layout of facilities so as to minimize the weighted sum of the distances between all pairs of facilities. The weights for each of the pairs of

*Corresponding author. Tel.:+91 7878088002

Email addresses: `ravikothari@iimahd.ernet.in` (Ravi Kothari), `diptesh@iimahd.ernet.in` (Diptesh Ghosh)

facilities as well as the lengths of each of the facilities are known beforehand, and the distance between a pair of facilities is defined as the distance between their centroids. The number of facilities in a particular instance is called the size of the instance. This problem was first proposed in Simmons (1969) and was shown to be NP-Hard in Beghin-Picavet and Hansen (1982). Formally stated the objective of SRFLP is to minimize the cost expression

$$z = \sum_{1 \leq i < j \leq n} c_{ij} d_{ij},$$

where c_{ij} is the weight and d_{ij} is the distance between the pair of facilities (i, j). Since the distance d_{ij} between the pairs of facilities depends on the linear arrangement of the facilities, the solution to the SRFLP is a permutation of facilities that minimizes the above cost expression.

The SRFLP has been used to model numerous practical situations. It has been used as a model for arrangement of rooms in hospitals, departments in office buildings or in supermarkets (Simmons 1969), arrangement of machines in flexible manufacturing systems (Heragu and Kusiak 1988), assignment of files to disk cylinders in computer storage, and design of warehouse layouts (Picard and Queyranne 1981). There are also large number of applications of the special case of the SRFLP in which the facilities have equal lengths. These include the triangulation problem of input output tables in economics (Laguna et al. 1999), and ranking of teams in sports (Martí and Reinelt 2011).

In the literature, several exact methods have been applied to solve the SRFLP to optimality. These methods include branch and bound (Simmons 1969), mathematical programming (Love and Wong 1976, Heragu and Kusiak 1991, Amaral 2006; 2008), cutting planes (Amaral 2009), dynamic programming (Picard and Queyranne 1981, Kouvelis and Chiang 1996), branch and cut (Amaral and Letchford 2012), and semidefinite programming (Anjos et al. 2005, Anjos and Vannelli 2008, Anjos and Yen 2009, Hungerländer and

Rendl 2011). These methods have been able to obtain optimal solutions to SRFLP instances with up to 42 facilities.

Researchers have focused on heuristics for solving larger sized SRFLP instances. These heuristics are of two types; construction and improvement. Construction heuristics for the SRFLP have been presented in Heragu and Kusiak (1988), Kumar et al. (1995), and Braglia (1997). However these have later been superseded by improvement heuristics. Most improvement heuristics for the SRFLP are metaheuristics, e.g. simulated annealing (Romero and Sánchez-Flores 1990, Kouvelis and Chiang 1992, Heragu and Alfa 1992), ant colony optimization (Solimanpur et al. 2005), scatter search (Kumar et al. 2008), tabu search (Samarghandi and Eshghi 2010), particle swarm optimization (Samarghandi et al. 2010), and genetic algorithms (Datta et al. 2011). Among these the genetic algorithm implementation in Datta et al. (2011) yield best results for benchmark SRFLP instances of large sizes.

The only application of genetic algorithm to the SRFLP is in Datta et al. (2011). In that paper, the authors demonstrate that genetic algorithms using basic genetic operators can yield low cost solutions to large SRFLP instances. In this work, we examine a large number of options available for the basic genetic operators and finally present a genetic algorithm which is competitive for large sized SRFLP instances. The largest sized instance reported in Datta et al. (2011) has 80 facilities, while in the current work, we examine instances with size up to 110.

Our paper is organized as follows. In Section 2 we introduce our genetic algorithm GA-KG for solving large instances of the SRFLP. We then report the results of our computational experiments with this genetic algorithm in Section 3 and compare our results with those available in the published literature. We conclude the paper in Section 4 with a summary of the work.

2. Our genetic algorithm

A genetic algorithm (see, e.g., Goldberg 1989) is an evolutionary search algorithm which simulates the natural evolution based on the principle of the survival of the fittest. An important component of a genetic algorithm is to map the phenotypic space to the genotypic space and design the relevant genetic operators to simulate the process of evolution. This mapping to the genotypic space essentially decides the representation of the solutions for the problem under consideration. The representations that have been widely used in the genetic algorithm literature are binary representation, real representation, integer representation and the permutation representation.

A genetic algorithm begins with an initial population consisting of a set of solutions (also known as the individuals or chromosomes) and evolves it over several generations by application of genetic operators such as selection, crossover, and mutation. The selection operator selects good quality solutions from the population to form a mating pool. The quality of a solution is generally decided by the value of a fitness function. In optimization problems with a minimization objective like the SRFLP, the fitness function is typically the inverse of the objective function value of the solution. The members of the mating pool have comparatively high fitness function values and are subsequently used for the crossover operation. The crossover operator generates child solutions by crossing parent solutions selected from the mating pool with a probability of crossover p_c. The children generated are then subjected to the mutation operator which creates slight perturbations with a very small probability of mutation p_m. After every generation, the algorithm uses an elite preserving mechanism which ensures that only the good quality solutions move to the next generation for the evolution. The weak solutions are eliminated with the hope that the fitter solutions in a particular generation will generate better solutions in subsequent generations and finally result in a good quality solution for the problem under consideration. This process of evolution of generation continues till a stopping criterion is reached.

In the remainder of this section we describe the components of our genetic algorithm implementation that we use for solving large sized SRFLP instances.

2.1. Solution representation

The representation of solutions in a genetic algorithm to solve a SRFLP is natural, since the solutions are simply permutations of facilities. So we use a permutation representation of solutions in our genetic algorithm.

2.2. Generation of the initial population

A population generation method for a genetic algorithm creates an initial generation of solutions which are then modified by the algorithm. There are two considerations while generating this initial population. The fitness of the solutions in the population should be high enough, and the solutions themselves should be diverse in the genotypic space to prevent premature convergence. Population generation methods are usually tailored to the problem that is being solved. For the SRFLP, we present four methods to generate solutions in the initial population. These methods have also been used in Datta et al. (2011).

Random initialization (RND). In this method, a solution generated is simply a random permutation of the set of all facilities that define the problem.

Worst pair together (WPT). In this method, a solution is obtained by arranging facilities into multiple subsequences, and finally merging the subsequences to obtain a permutation of facilities. Initially, there are no subsequences, and all pairs (i, j) of facilities are ordered in non-increasing values of $c_{ij}(l_i + l_j)/2$. Pairs of facilities are considered for adding to subsequences in this order. If both the facilities in the pair being considered are parts of existing subsequences, then that pair is ignored. If one of the two facilities is a part of an existing subsequence, then the other is located at one of the two ends of that subsequence with the objective of minimizing the cost of the subsequence. If neither of the two facilities belong to any existing subsequence, then the pair forms a new subsequence and the next pair in the ordering is considered. After all the pairs have been considered, all the remaining subsequences are joined to the first subsequence obtained at the start of the process, one at a time with the objective of achieving a lowest cost ordering every time a new subsequence gets joined. Note that this method is deterministic.

Flow based permutation (FBP). In this method, the facilities are considered in non-decreasing order of their average weight values $w_i = \sum_{j=1}^{n} c_{ij} l_i$. A permutation of facilities is obtained by placing the first two facilities next to each other, and then placing the j-th facility, $j = 3, \ldots, n$, in the ordering on the unoccupied side of the $(j-2)$-th facility. This method is also deterministic.

Length based permutation (LBP). In this method, the facilities are considered in non-decreasing order of their lengths. The first two facilities in this ordering are placed next to each other. The j-th facility, $j = 3, \ldots, n$, in the ordering is placed on the unoccupied side of the $(j-2)$-th facility. This method is again a deterministic method.

Our initial experiments showed that the solutions generated by the RND method typically had significantly higher costs (and hence lower fitness) than those generated by the other three methods. In the genetic algorithm described in Datta et al. (2011), in order to generate a population of size N, $N/2$ solutions are generated using the RND method and each of the other $N/2$ solutions is generated by randomly choosing one among WBT, FBP, and LBP methods. This was possibly done to ensure that solutions generated by the WBT, FBP, and LBP methods were chosen in the mating pool for the next generation. However, since the fitness of solutions generated by these three methods was much higher than those generated by the RND method, their selection was almost certain even if the population did not contain too many copies of these solutions. On the other hand, reducing the number of solutions that were generated by these three methods allows us to generate more solutions by the RND method, which in turn increases the diversity of the initial population. In our genetic algorithm implementation therefore, we generate $N-3$ solutions using the RND method and one each using WBT, FBP, and LBP methods. Measuring the diversity of populations using the deviation distance measure (see, e.g., Sörensen 2007) we found that the diversity of our initial population was 10.3% higher on average than that in Datta et al. (2011).

2.3. Creation of a mating pool

Given a generation of solutions, the first step in creating the next generation is to create a mating pool for reproduction in a genetic algorithm. The solutions in the mating pool are subsequently subjected to crossover and mutation operations to create candidate solutions for the next generation. The solutions in a mating pool are selected from the population based on their fitness values. This may allow multiple copies of the same solution in the mating pool. Hence a mating pool encourages fitter solutions and eliminates the weaker ones. In general the size of the mating pool is same as that of the population.

Several methods of creating a mating pool have been described in the literature. However the following two methods have been most widely used.

Roulette wheel selection. In a roulette wheel selection method, solutions from a generation are chosen to be members of the mating pool with probabilities that are proportional to their fitness values. Hence this method is biased towards selecting solutions which have better objective function values, and ensures that bad quality solutions have a very low chance of being propagated across generations in the genetic algorithm.

Tournament selection. Tournament selection involves running several tournaments among k solutions chosen at random from a population. The winner of each tournament is selected to be a member of the mating pool. Generally, the fittest among the chosen solutions is the winner of a tournament. The parameter k is known as the size of the tournament. Another way of selecting solutions to enter the mating pool is to choose the fittest solution with a user-specified probability p, the second fittest solution with probability $p \cdot (1-p)$, the third fittest solution with probability $p \cdot (1-p)^2$, and so on. A tournament selection method that has been found to be very effective is the binary tournament selection. In this method two solutions are randomly selected from the population and the best solution based on the fitness value is selected for the mating pool. The process is repeated till the desired size of the mating pool is achieved.

Our initial experiments showed that the mating pool obtained using a roulette wheel selection method usually generates populations with higher average fitness but with lower diversity than a binary tournament selection method. Consequently, if the roulette selection method is used, the genetic algorithm converges prematurely and outputs solutions which are worse than those output when the binary tournament selection is used. In our genetic algorithm implementation therefore, we use the binary tournament selection operator to create a mating pool.

2.4. The crossover operation

The crossover operation mates solutions in the mating pool to create a population of solutions which, after mutation, will be candidates for selection into the next generation of solutions. Since the SRFLP is a problem in which a solution is a permutation of facilities, we restrict our choice of crossover operators among those which are specifically designed for such problems. The operators from which we select the crossover operator for our implementation of the genetic algorithm are the following.

Partially-matched crossover (PMX). In this crossover operation, two parent solutions from the mating pool generate a child solution. Two parent solutions P_1 and P_2 are chosen from the mating pool with a probability p_c. A random segment of facilities, called a crossover segment, is then selected from P_1 and copied to same location in the child solution. Let the segment consist of facilities from position s to position t in P_1. Next the facilities in P_2 from position s to position t which are not a part of the crossover segment are considered. Let π_i be such a facility, and let π_i^c be the facility from the crossover segment that occupies the same position in the child solution. Then π_i is placed in the same position that π_i^c occupies in P_2. The facilities that have not been assigned positions in the child solution so far are then assigned positions in the same order in which they appear in P_2.

Cycle crossover (CX). In this crossover operation, two parent solutions from the mating pool create a child solution. Two parent solutions P_1 and P_2 are chosen from the mating pool with a probability p_c. Facilities are first assigned to the child solution using a P_1 cycle. To do this, a facility π_i^1 is chosen from P_1 and added to the same location in the child solution. Let π_i^2 be the facility in P_2 at the same position. Then π_i^2 is added in the child solution in the same position that it occupies in P_1. Let this be the j-th position. Next the facility π_j^2 in the j-th position in P_2 is considered. This process continues until no further facilities can be added to the child solution. This completes the P_1 cycle. Facilities are then assigned to the child solution using a P_2 cycle which is similar to a P_1 cycle, but starting from a facility in P_2. The CX operation continues to alternate between P_1 and P_2 cycles until the child solution is complete.

Order 1 crossover (OX1). This technique starts with choosing two parent solutions P_1 and P_2 from the mating pool with a probability p_c. Then a sub-permutation of facilities is randomly selected from P_1 and is copied to the child solution at the same location. Let this sub-permutation occupy positions i through j. The remaining positions in the child solution are filled up by the facilities from P_2. To do this an ordering of facilities is created by appending the sequence of facilities from the first position to the j-th position in P_2 to the sequence of facilities from the $(j+1)$-th position to the n-th position in P_2. The empty positions in the child solution are then assigned facilities in this order starting from the $(j+1)$-th position in the child solution and wrapping around. While assigning the facilities, those which are already present in the child solution are omitted.

Order-based crossover(OX2). This technique begins by choosing two parent solutions P_1 and P_2 from the mating pool with a probability p_c. After this two random crossover points i and j are selected. The sub-permutations of P_1 from the first position to the i-th position and from the j-th position to the n-th position are copied to the same positions in the child solution. The remaining positions in the child solution are filled with facilities between the $(i+1)$-th and $(j-1)$-th positions in P_1 but in the order in which they occur in P_2.

We performed preliminary experiments with the four crossover operators on problems of size between 60 and 80. We used the initial population generation method and the selection operator which we have specified earlier in this section. Our experiments showed that the genetic algorithm using the PMX crossover operator generated the lowest cost solutions among the four. Hence in our genetic algorithm we use the PMX crossover operator.

2.5. The mutation operation

After generating the children from the crossover operator, each child is subjected to a mutation operator. In mutation, a solution is changed in a minor way with a small mutation probability p_m, thereby preventing a premature convergence of the algorithm to sub optimal solutions. We tested out the following two widely used mutation operators for permutation representations.

Insert Mutation. In this mutation, two facilities in the solution are selected at random with low probabilities p_m and then the second facility is moved to follow the first facility in the solution. All the remaining facilities are adequately shifted to finally obtain a mutated child solution.

Swap Mutation. In this mutation, two facilities are selected with a small probability p_m and their locations in the solution are interchanged to obtain a mutated child solution.

We performed preliminary experiments with the two mutation operators on problems of size between 60 and 80. We used the initial population generation method, the selection operator, and the crossover operator which we have specified earlier in this section. Our experiments showed that the genetic algorithm using the insertion mutation operator generated distinctly better solutions than those generated using the swap mutation. Hence in our genetic algorithm we used the insertion mutation operator.

2.6. Elite preservation

In a genetic algorithm incorporating only selection, crossover, and mutation operations, solutions of one generation are almost never carried over to the next generation. This is a weakness of the algorithm, since a good solution in a generation is capable of generating other good solutions in subsequent generations as long as the average solution fitness in those generations are not significantly superior to the fitness of the solution. In order to overcome this weakness, genetic algorithms use an elite preservation operation in which good solutions in a generation are copied to the next generation.

In our genetic algorithm implementation, we use an elite preservation operation in which we pool together the solutions from the previous generation and all the mutated children, and finally choose the best N solutions from this pool to form the next generation. This is identical to the elite preservation operation used in Datta et al. (2011).

In summary, the genetic algorithm that we present in this paper, which we call GA-KG, uses three parameters; population size (N), crossover probability (p_c), and mutation probability (p_m) and has the following components:

1. Initial population generation with $N-3$ solutions using the RND method, and one each using WBT, FBP, and LBP methods;
2. Mating pool generation using binary tournament selection.
3. Crossover using the PMX crossover operator;
4. Mutation using the insertion mutation operator; and
5. Elite preservation which chooses the best N solutions from a combined pool of the solutions in the previous generation and the mutated child solutions.

In the next section we describe the results of our computational experience with GA-KG on large sized SRFLP instances.

3. Computational experience

We coded our genetic algorithm in C and compiled it using a `gcc4` compiler. We ran our experiments on a personal computer with Intel i-5 2500 3.30 GHz processor with 4GB RAM running Ubuntu Linux version 11.10. Based on preliminary experiments we chose to implement GA-KG with a population size $N = 60$. Following the implementation in Datta et al. (2011) we allowed the crossover probability p_c to be randomly chosen among values 0.6, 0.7, 0.8, 0.9, and 0.95, and the mutation probability p_m to be randomly chosen between 0.001 and 0.05. Each run of GA-KG was allowed to generate a maximum of 5000 generations or till the population converged. Since GA-KG is stochastic, we performed 200 GA-KG runs and chose the best solution obtained as the output of GA-KG.

We compared the performance of GA-KG with results from the published literature on three classes of benchmark instances. The first class of 20 instances are the Anjos instances used in Anjos and Yen (2009) and subsequently by other researchers. This class contains five instances each of size

60, 70, 75, and 80. The second class of 20 instances was used in (Anjos and Yen 2009) and in Hungerländer and Rendl (2011). It consists of instances based on QAP benchmarks and are called `sko` instances. There are five instances each of sizes 64, 72, 81, and 100. The third class is a set of three instances of size 110 considered in Amaral and Letchford (2012).

Table 1 presents a comparison of the results obtained by GA-KG with those known from the literature for Anjos instances. The first two columns of the table give the name of the instance and its size. The third column reports the costs of the best solutions known for the instance in the literature. The fourth column presents the cost of the best solution for this problem obtained by the genetic algorithm implementation in Datta et al. (2011). We include this since it is the only other genetic algorithm for the SRFLP to the best of our knowledge. The last column reports the cost of the best solution obtained by GA-KG. The figures marked in boldface in Table 1 indicate the five Anjos instances in which GA-KG generates the solutions with costs lower than those previously known in the literature. In the other 15 instances, it matches the best known results from the published literature. GA-KG produces better solutions than the ones reported in Datta et al. (2011) in 11 of the 20 Anjos instances. The average time per run of the GA-KG implementation is 6.3 seconds, 10.0 seconds, 12.5 seconds, and 16.6 seconds for instances of size 60, 70, 75, and 80 respectively. The layouts for the instances in which we have improved the previously known best solutions are provided in the Appendix.

Table 2 presents a comparison of the results obtained by GA-KG with those known from the literature for the QAP based `sko` instances. The first two columns of the table give the name of the instance and its size. The third column reports the costs of the best solutions known from Hungerländer and Rendl (2011). We tested the implementation in Datta et al. (2011) on these instances and report the costs of the best solutions obtained from the implementation in the third column. The last column of the table reports the cost of the best solutions obtained by GA-KG. Here too the figures marked

Table 1: Results for Anjos instances of sizes from 60 to 80

Instance	Size	Best[a]	DA&F[b]	GA-KG
Anjos-60-01	60	1477834.0	1477834.0	1477834.0
Anjos-60-02	60	841792.0	841792.0	841776.0
Anjos-60-03	60	648337.5	648337.5	648337.5
Anjos-60-04	60	398468.0	398511.0	398406.0
Anjos-60-05	60	318805.0	318805.0	318805.0
Anjos-70-01	70	1528621.0	1529197.0	**1528537.0**
Anjos-70-02	70	1441028.0	1441028.0	1441028.0
Anjos-70-03	70	1518993.5	1518993.5	1518993.5
Anjos-70-04	70	968796.0	969130.0	968796.0
Anjos-70-05	70	4218017.5	4218230.0	4218002.5
Anjos-75-01	75	2393456.5	2393483.5	2393456.5
Anjos-75-02	75	4321190.0	4321190.0	4321190.0
Anjos-75-03	75	1248537.0	1248551.0	**1248423.0**
Anjos-75-04	75	3941981.5	3942013.0	**3941816.5**
Anjos-75-05	75	1791408.0	1791408.0	1791408.0
Anjos-80-01	80	2069097.5	2069097.5	2069097.5
Anjos-80-02	80	1921177.0	1921177.0	**1921136.0**
Anjos-80-03	80	3251368.0	3251413.0	3251368.0
Anjos-80-04	80	3746515.0	3746515.0	3746515.0
Anjos-80-05	80	1588901.0	1589061.0	**1588885.0**

a: Best refers to the best solutions among Datta et al. (2011) and upper bounds obtained in Hungerländer and Rendl (2011)

b: DA&F refers to Datta et al. (2011)

in boldface in Table 1 indicate the `sko` instances in which GA-KG generates the solutions with costs lower than those previously known in the literature. GA-KG outperforms the previously best known results in 17 out of the 20 instances. In two instances the cost of the best solution obtained by GA-KG equals those obtained previously in the literature. In the remaining instance

Table 2: Results for `sko` instances of sizes from 64 to 100

Instance	Size	H&R[a]	A&L[b]	GA-DA&F[c]	GA-KG
`sko-64-01`	64	97194.0	96930.0	97218.0	97057.0
`sko-64-02`	64	634332.5	634332.5	634345.5	634332.5
`sko-64-03`	64	414384.5	414356.5	414323.5	**414323.5**
`sko-64-04`	64	298155.0	297358.0	297532.0	**297261.0**
`sko-64-05`	64	502063.5	501922.5	502300.5	501922.5
`sko-72-01`	72	139231.0	139174.0	139191.0	**139150.0**
`sko-72-02`	72	715611.0	712261.0	712370.0	**712253.0**
`sko-72-03`	72	1061762.5	1054184.5	1055400.5	**1054110.5**
`sko-72-04`	72	924019.5	920693.5	919841.5	**919590.5**
`sko-72-05`	72	430288.5	428305.5	428954.5	**428228.5**
`sko-81-01`	81	207063.0	205475.0	**205341.0**	205413.0
`sko-81-02`	81	526157.5	523021.5	521391.5	**521391.5**
`sko-81-03`	81	979281.0	970920.0	971797.0	**970897.0**
`sko-81-04`	81	2035569.0	2032634.0	2032045.0	**2031803.0**
`sko-81-05`	81	1311166.0	1303756.0	1303327.0	**1302733.0**
`sko-100-01`	100	380562.0	378584.0	378697.0	**378378.0**
`sko-100-02`	100	2084924.5	2076714.5	2079182.5	**2076037.5**
`sko-100-03`	100	16216076.5	16177226.5	16155456	**16160222.0**
`sko-100-04`	100	3263493.0	3237111.0	3242000.0	**3233197.0**
`sko-100-05`	100	1040929.5	1034922.5	1033452.5	**1033356.5**

a: H&R refers to upper bounds obtained inHungerländer and Rendl (2011)
b: A&L refers to upper bounds obtained inAmaral and Letchford (2012)
c: GA-DA&F refers to our implementation using the same method as in Datta et al. (2011)

the best solutions were those reported in Amaral and Letchford (2012). The average time per run of the GA-KG implementation is 7.3 seconds, 10.9 seconds, 16.9 seconds, and 34.2 seconds for instances of size 64, 72, 81, and 100 respectively. The layouts for the instances in which we have improved the previously known best solutions are provided in the Appendix.

In the third set of benchmark problems considered, GA-KG using the parameters specified earlier was able to generate good solutions in one of the three instances. However, when the population size was increased to $N = 200$ and the maximum number of generations to 10000, we obtained results that were better than those published in Amaral and Letchford (2012) in two of the three instances. The costs of the best solutions that GA-KG output for these problems is presented in Table 3. The first four columns provide the details of the instances, the fifth column reports the costs of best solutions in Amaral and Letchford (2012) and the last column presents the costs of the best solutions obtained by GA-KG. The figures marked in boldface indicate the instances in which GA-KG produced better results than is known in the literature. The average time per run of the GA-KG implementation is 68.0

Table 3: Results for Amaral instances of size 110

Instance	Size	l-range	c-range	A&L[a]	GA-KG
Amaral-110-01	110	[7,23]	[0,100]	144331884.5	**144302160.0**
Amaral-110-02	110	[5,14]	[0,100]	86065390.0	**86056632.0**
Amaral-110-03	110	[3, 7]	[0,5]	2234803.5	2234876.5

a: A&L refers to upper bounds obtained in Amaral and Letchford (2012)

seconds for these instances when $N = 60$ and 111.2 seconds when $N = 200$. The layouts for the instances in which we have improved the previously known best solutions are provided in the Appendix.

From our computational experiments it is clear that the GA-KG implementation is superior to other implementations known in the literature.

4. Summary

In this paper, we propose a genetic algorithm called GA-KG for solving large instances of the single row facility layout problem (SRFLP). Our implementation generates an initial population using a combination of four approaches to generate feasible solutions to the SRFLP. It then creates successive generations in four steps. In the first step a mating pool of solutions is created using the binary tournament selection operator. In the second step, solutions from the mating pool are selected to generate child solutions through a PMX crossover mechanism. In the third step, children thus generated are subjected to an insertion mutation operation. Finally the new generation is created using an elite preservation operation.

We compared the performance of GA-KG with that of other algorithms known in the literature for solving large sized SRFLP instances. Our test bed consisted of 40 instances with sizes ranging from 60 to 110. We found that GA-KG output solutions that matched the best known solutions in 17 of the 43 instances. In 24 instances, it produced solutions which are superior to the best solutions known in the literature. In the other two instances, the solutions output by GA-KG were marginally worse than the best solutions known in the literature. The times required per run of the proposed algorithm are comparable to the fastest algorithms known in the literature for these problems. Hence the GA-KG algorithm that we propose in this paper is an algorithm of choice to solve large sized single row facility layout problems.

References

Amaral, A. and Letchford, A. N. (2012). A polyhedral approach to the single row facility layout problem. Available at http://www.lancs.ac.uk/staff/letchfoa/articles/SRFLP-rev.pdf.

Amaral, A. R. S. (2006). On the exact solution of a facility layout problem. *European Journal of Operational Research*, 173(2):508–518.

Amaral, A. R. S. (2008). An Exact Approach to the One-Dimensional Facility Layout Problem. *Operations Research*, 56(4):1026–1033.

Amaral, A. R. S. (2009). A new lower bound for the single row facility layout problem. *Discrete Applied Mathematics*, 157(1):183–190.

Anjos, M., Kennings, a., and Vannelli, a. (2005). A semidefinite optimization approach for the single-row layout problem with unequal dimensions. *Discrete Optimization*, 2(2):113–122.

Anjos, M. F. and Vannelli, A. (2008). Computing Globally Optimal Solutions for Single-Row Layout Problems Using Semidefinite Programming and Cutting Planes. *INFORMS Journal on Computing*, 20(4):611–617.

Anjos, M. F. and Yen, G. (2009). Provably near-optimal solutions for very large single-row facility layout problems. *Optimization Methods and Software*, 24(4-5):805–817.

Beghin-Picavet, M. and Hansen, P. (1982). Deux problèmes daffectation non linéaires. *RAIRO, Recherche Opérationnelle*, 16(3):263–276.

Braglia, M. (1997). Heuristics for single-row layout problems in flexible manufacturing systems. *Production Planning & Control*, 8(6):558–567.

Datta, D., Amaral, A. R., and Figueira, J. R. (2011). Single row facility layout problem using a permutation-based genetic algorithm. *European Journal of Operational Research*, 213(2):388–394.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

Heragu, S. S. and Alfa, A. S. (1992). Experimental analysis of simulated annealing based algorithms for the layout problem. *European Journal of Operational Research*, 57(2):190–202.

Heragu, S. S. and Kusiak, A. (1988). Machine Layout Problem in Flexible Manufacturing Systems. *Operations Research*, 36(2):258–268.

Heragu, S. S. and Kusiak, A. (1991). Efficient models for the facility layout problem. *European Journal Of Operational Research*, 53:1–13.

Hungerländer, P. and Rendl, F. (Unpublished results, 2011). A computational study for the single-row facility layout problem. Available at `www.optimization-online.org/DB_FILE/2011/05/3029.pdf`.

Kouvelis, P. and Chiang, W.-C. (1992). A simulated annealing procedure for single row layout problems in flexible manufacturing systems. *International Journal of Production Research*, 30(4):717–732.

Kouvelis, P. and Chiang, W.-C. (1996). Optimal and Heuristic Procedures for Row Layout Problems in Automated Manufacturing Systems. *Journal of the Operational Research Society*, 47(6):803–816.

Kumar, R. K., Hadejinicola, G. C., and Lin, T.-L. (1995). A heuristic procedure for the single-row facility layout problem. *European Journal of Operational Research*, 87(1):65–73.

Kumar, S., Asokan, P., Kumanan, S., and Varma, B. (2008). Scatter search algorithm for single row layout problem in fms. *Advances in Production Engineering & Management*, 3(4):193–204.

Laguna, M., Martí, R., and Campos, V. (1999). Intensification and diversification with elite tabu search solutions for the linear ordering problem. *Computers & OR*, 26(12):1217–1230.

Love, R. F. and Wong, J. Y. (1976). On solving a one-dimensional space allocation problem with integer programming. *INFOR*, 14(2):139–144.

Martí, R. and Reinelt, G. (2011). *The Linear Ordering Problem*. Springer-Verlag Berlin Heidelberg.

Picard, J.-C. and Queyranne, M. (1981). On the one-dimensional space allocation problem. *Operations Research*, 29(2):371–391.

Romero, D. and Sánchez-Flores, A. (1990). Methods for the one-dimensional space allocation problem. *Computers & Operations Research*, 17(5):465–473.

Samarghandi, H. and Eshghi, K. (2010). An efficient tabu algorithm for the single row facility layout problem. *European Journal of Operational Research*, 205(1):98–105.

Samarghandi, H., Taabayan, P., and Jahantigh, F. F. (2010). A particle swarm optimization for the single row facility layout problem. *Computers & Industrial Engineering*, 58(4):529–534.

Simmons, D. M. (1969). One-Dimensional Space Allocation: An Ordering Algorithm. *Operations Research*, 17(5):812–826.

Solimanpur, M., Vrat, P., and Shanker, R. (2005). An ant algorithm for the single row layout problem in flexible manufacturing systems. *Computers & Operations Research*, 32(3):583–598.

Sörensen, K. (2007). Distance measures based on the edit distance for permutation-type representations. *Journal of Heuristics*, 13:35–47. 10.1007/s10732-006-9001-3.

Appendix

We provide details of the solutions for the instances in which we have improved the best solution known in the literature. Note that the facilities are numbered from 0 through $n - 1$ where n is the problem size.

Instance	Size	Cost	Permutation
Anjos-70-01	70	1528537.0	52 46 64 1 39 27 61 21 14 7 31 8 62 30 68 50 67 0 3 15 63 60 40 37 55 66 69 43 9 25 13 18 32 41 48 4 29 35 22 54 59 12 17 20 23 26 53 10 11 57 5 58 51 6 19 65 2 33 44 45 24 42 47 16 28 56 38 34 36 49
Anjos-75-03	75	1248423.0	46 68 41 9 18 32 14 16 42 50 40 45 28 22 67 25 59 3 38 73 63 60 55 19 35 11 26 12 47 70 10 64 56 4 66 44 20 27 34 23 8 74 57 72 39 6 31 5 48 51 58 33 2 15 61 30 29 43 36 1 37 65 69 17 71 7 24 54 52 62 13 0 53 49 21
Anjos-75-04	75	3941816.0	35 59 4 13 14 49 6 74 9 41 61 36 7 69 29 46 21 56 19 40 28 39 32 38 45 11 2 63 34 64 15 51 27 52 43 72 33 17 23 44 12 31 0 66 1 18 54 47 55 62 65 25 22 57 58 53 42 70 3 30 10 73 60 50 5 24 26 67 68 37 71 48 8 16 20
Anjos-80-02	80	1921136.0	21 56 17 23 9 16 54 4 57 13 45 14 55 31 73 46 11 53 40 28 15 32 76 0 71 26 27 22 24 34 20 5 66 2 52 19 3 37 38 35 8 39 33 7 64 1 49 43 61 41 74 12 25 67 18 58 75 72 29 69 6 78 36 70 30 51 63 77 59 79 50 68 62 48 42 47 44 60 65 10

Instance	Size	Cost	Permutation
Anjos-80-05	80	1588885.0	24 16 47 40 57 59 68 77 12 48 41 71 11 28 15 50 13 63 25 27 23 14 55 4 56 49 45 22 62 39 44 19 79 3 17 76 38 31 2 58 64 54 65 70 34 52 42 5 43 53 74 78 33 29 61 35 32 75 8 18 30 7 69 0 21 10 73 36 9 67 66 20 72 60 26 37 51 46 6 1
sko-64-03	64	414323.5	14 11 60 8 55 40 41 48 12 28 3 51 21 22 15 45 35 50 63 54 20 26 30 2 43 13 57 56 23 52 9 24 62 42 17 46 29 34 16 37 33 44 0 38 4 59 25 27 39 10 53 1 7 32 36 18 31 47 19 6 49 58 61 5
sko-64-04	64	297261.0	30 59 41 35 42 26 62 5 17 19 45 50 2 51 60 39 13 47 27 8 21 3 23 11 61 38 54 28 4 63 15 12 48 57 46 43 20 34 22 44 6 0 36 33 25 49 37 1 16 24 55 32 9 18 7 29 40 31 53 10 52 56 58 14
sko-64-05	64	501922.5	35 8 17 21 28 62 48 19 45 60 11 51 13 9 63 12 57 54 41 15 5 4 42 50 46 30 26 3 2 22 39 27 59 47 23 29 55 20 34 44 40 53 10 38 32 33 37 31 7 6 56 36 0 43 1 18 16 25 49 52 24 14 58 61
sko-72-01	72	139150.0	11 52 30 63 59 17 34 7 55 9 69 26 29 16 66 20 1 13 21 27 25 46 64 51 22 60 28 0 6 37 58 45 48 32 49 12 36 62 31 38 15 8 5 70 43 54 56 35 50 67 41 39 47 40 14 18 57 19 65 42 3 61 71 68 24 4 10 44 53 23 2 33
sko-72-02	72	712253.0	11 58 0 31 13 12 22 51 46 36 62 64 52 2 42 27 69 21 34 9 32 45 6 59 48 37 1 26 29 49 20 30 24 28 7 66 50 41 54 43 25 23 56 15 38 68 35 67 60 3 65 71 14 5 47 18 53 16 70 39 8 55 19 44 10 63 61 17 40 57 4 33

Instance	Size	Cost	Permutation
sko-72-03	72	1054110.5	30 13 55 59 17 38 16 66 8 27 25 29 9 69 58 60 61 20 7 49 45 12 62 64 21 63 22 47 32 53 26 15 34 46 52 5 42 36 11 24 68 51 41 23 67 18 65 31 10 44 39 56 43 28 35 3 0 33 70 19 54 40 50 37 48 1 2 4 14 71 6 57
sko-72-04	72	919590.5	11 2 55 63 23 71 44 49 35 40 19 10 64 60 14 3 65 42 61 62 36 57 33 68 24 5 52 17 4 18 38 8 67 66 15 53 47 50 41 39 43 56 54 30 37 59 28 45 22 48 21 13 34 32 70 69 12 9 29 6 26 27 0 46 51 25 20 1 31 58 16 7
sko-72-05	72	428228.5	50 28 33 39 70 8 43 22 0 6 32 37 54 15 40 57 58 45 64 5 18 14 19 13 48 65 67 60 35 3 56 4 44 68 24 12 23 10 42 25 31 46 51 41 61 62 21 49 36 20 11 1 34 71 16 30 66 63 59 38 47 29 27 53 26 69 9 17 55 2 52 7
sko-81-01	81	205341.0	7 46 59 53 52 9 39 70 27 71 45 15 75 63 76 56 19 23 28 54 21 42 74 58 72 0 61 77 67 26 44 12 51 57 62 36 17 6 50 4 78 41 31 11 43 1 10 79 38 34 69 35 30 37 48 18 14 22 40 13 3 5 80 20 25 16 55 32 47 8 33 66 64 68 49 24 29 60 2 65 73
sko-81-02	81	521391.5	7 27 76 59 63 56 75 15 46 71 53 61 6 62 26 9 38 19 17 23 42 52 74 70 39 58 72 36 67 12 45 50 77 0 57 4 78 28 79 31 11 44 1 43 18 30 25 33 8 10 3 41 64 29 35 66 22 49 40 69 73 65 32 16 68 2 20 60 14 5 55 47 51 13 80 34 24 48 37 54 21

Instance	Size	Cost	Permutation
sko-81-03	81	970897.0	46 76 52 15 43 53 9 33 39 75 59 58 62 19 36 50 12 17 63 67 56 20 26 61 23 77 42 6 48 21 72 57 68 74 51 44 0 7 70 31 45 55 3 66 41 22 16 49 64 40 25 80 65 35 79 18 11 32 37 69 2 24 60 1 13 14 30 8 4 5 47 71 27 78 10 28 54 73 38 34 29
sko-81-04	81	2031803.0	73 65 7 24 25 64 68 22 50 2 13 80 19 49 1 29 60 20 11 27 5 43 8 10 18 16 14 69 47 32 45 38 71 39 33 31 48 59 3 66 36 55 77 41 53 9 17 34 35 54 0 44 51 26 61 57 74 63 46 62 72 30 12 67 56 75 76 23 52 37 15 42 58 6 40 4 28 21 78 70 79
sko-81-05	81	1302733.0	39 33 53 78 3 21 63 28 37 46 74 64 29 79 34 25 41 59 42 18 17 11 62 36 75 22 76 68 32 72 58 50 23 66 57 19 24 9 48 70 52 13 40 27 80 31 10 45 0 49 16 20 43 15 12 30 56 69 77 38 61 6 44 1 26 67 60 71 54 2 4 8 7 14 5 35 51 47 55 73 65
sko-100-01	100	378378.0	64 77 17 96 0 36 70 82 38 78 90 32 57 9 94 24 59 10 12 26 4 62 56 31 76 5 14 27 87 73 54 53 79 74 33 30 15 21 61 51 85 86 83 8 58 18 41 97 3 55 95 92 67 7 88 23 72 29 84 66 93 98 42 13 39 19 68 99 25 45 60 65 46 89 63 50 80 1 48 69 71 37 91 81 22 47 40 49 35 11 44 52 75 6 20 16 28 43 34 2
sko-100-02	100	2076037.5	32 28 16 98 2 52 11 60 76 40 35 75 47 49 93 37 71 6 25 39 67 7 80 42 13 58 99 69 68 19 45 34 66 44 91 48 50 51 20 1 97 46 65 89 43 85 18 0 81 22 3 55 54 27 14 56 88 5 33 79 73 86 36 15 21 30 78 83 74 12 82 70 62 53 61 95 29 84 8 63 59 94 10 31 24 87 72 17 92 9 64 41 23 38 4 26 57 96 90 77

Instance	Size	Cost	Permutation
sko-100-03	100	16160222.0	96 57 27 75 59 48 44 52 91 60 46 40 39 99 68 1 69 89 42 65 28 86 64 72 9 71 63 29 84 95 50 17 47 6 11 22 37 25 55 92 41 58 16 83 3 87 24 31 21 76 80 19 97 33 56 78 73 94 54 0 45 5 49 66 35 51 14 98 85 93 18 13 10 15 8 61 67 82 12 74 26 62 90 81 30 70 36 4 20 53 79 23 88 32 38 7 77 43 34 2
sko-100-04	100	3233197.0	6 2 5 83 44 57 75 1 98 16 11 22 28 43 94 63 33 86 40 13 72 46 29 19 39 9 50 35 69 27 65 71 52 34 68 81 25 37 99 89 45 47 91 60 49 80 76 58 90 56 36 61 84 62 77 54 12 10 3 64 95 14 55 88 0 53 30 73 18 85 66 59 23 79 74 26 20 42 32 21 87 15 8 82 97 7 67 51 93 4 17 96 92 31 24 41 70 78 38 48
sko-100-05	100	1033356.5	77 75 2 49 47 44 29 11 57 9 59 64 61 22 84 95 3 27 73 81 53 35 20 38 23 79 34 74 6 60 36 70 82 15 87 41 8 69 26 96 94 1 92 78 52 17 4 90 63 19 45 56 33 40 25 98 31 0 80 13 86 67 66 32 93 72 65 39 10 55 28 43 30 88 21 46 99 7 76 42 48 97 14 18 83 37 58 50 68 91 51 5 16 24 12 62 71 85 54 89
Amaral-110-01	110	144302160.0	39 18 61 72 74 66 64 8 108 10 75 2 85 109 32 3 107 91 13 6 78 54 21 23 38 92 0 68 83 55 14 27 15 4 63 73 76 70 90 49 25 89 45 96 98 87 41 42 34 24 46 101 50 19 94 95 59 20 62 86 57 9 81 84 35 71 26 36 100 103 102 67 65 44 37 58 60 80 88 104 11 30 31 69 12 77 106 29 33 17 22 40 105 93 28 51 43 1 47 56 99 53 16 48 5 97 82 52 79 7

Instance	Size	Cost	Permutation
Amaral-110-02	110	86056632.0	63 61 0 12 32 10 103 68 73 24 53 15 4 56 85 54 46 95 33 79 13 57 2 69 100 90 74 47 50 45 8 98 55 25 14 87 38 75 67 1 83 106 29 99 41 21 92 27 6 34 76 91 72 70 107 22 3 89 84 93 5 82 77 51 71 81 59 23 48 9 96 16 18 30 62 52 42 65 44 37 60 40 28 19 26 88 43 104 94 80 78 109 49 35 105 101 58 108 97 36 64 102 17 20 66 86 31 39 11 7