Population Heuristics for the Corridor Allocation Problem

Diptesh Ghosh
Ravi Kothari

W.P. No. 2012-09-02
September 2012

INDIAN INSTITUTE OF MANAGEMENT
AHMEDABAD – 380015
INDIA

Population Heuristics for the Corridor Allocation Problem

Diptesh Ghosh
Ravi Kothari

Abstract

The corridor allocation problem is one of assigning a given set of facilities in two rows along a straight corridor so as to minimize a weighted sum of the distances between every pair of facilities. This problem has practical applications in arrangements of rooms in offices and in hospitals. The problem is NP-hard. In this paper, we present two population based metaheuristic implementations for the problem; a genetic algorithm with local search embedded in it, and a scatter search algorithm with path relinking. We report the results of our experiments with these algorithms on benchmark instances with up to 49 facilities.
Keywords: Facility planning and design, corridor allocation, double row layout, genetic algorithm, scatter search, path relinking.

1 Introduction

Consider a set $F = \{f_1, f_2, \ldots, f_n\}$ of $n \geq 2$ facilities, where the length of facility f_j is l_j. Also consider a non-negative weight c_{ij} for each pair $\{f_i, f_j\}$ of facilities in F, which represents the communication cost between the pair. The corridor allocation problem (CAP; see Amaral 2012) seeks to arrange the facilities in F in two horizontal rows so as to minimize the total cost of communication among the facilities. The arrangement must follow two rules; (i) there can be no gap between two consecutive facilities in any row; and (ii) the left edges of the left most facilities in each of the two rows has to be at a horizontal distance of 0 from the origin. Thus, in Figure 1, (A) represents a solution to the CAP, while (B) and (C) do not, since (B) violates condition (i) and (C) violates condition (ii). The cost of an arrangement of facilities is the weighted sum of the distances between the centroids of every pair of facilities. Thus if in an arrangement S of facilities, the horizontal coordinate of the centroid of facility f_i is denoted as x_i^S, then the cost $z(S)$ of the arrangement S is

$$z(S) = \sum_{f_i, f_j \in F} c_{ij} |x_i^S - x_j^S|. \tag{1}$$

This NP-hard problem finds applications in areas such as room arrangements in office buildings and in hospitals (Amaral 2012).

Formally an instance of CAP is represented using the ordered set (F, L, C) where F is the set of facilities, $L = (l_j)$ is a vector of facility lengths, and $C = [c_{ij}]$ is a symmetric matrix of weights. In this paper, we assume that each row in C matrix has at least one non-zero element. The assumption is not restrictive; if the i-th row has no non-zero element, then an optimal solution to the CAP (F, L, C) can be obtained by finding an optimal solution to the CAP obtained by removing facility f_i from F, and finally adding f_i to the extreme right in any of the two rows.

The CAP is part of a group of associated facility layout problems. Other members of this group are the single row facility layout problem (SRFLP) in which the objective is to arrange facilities in

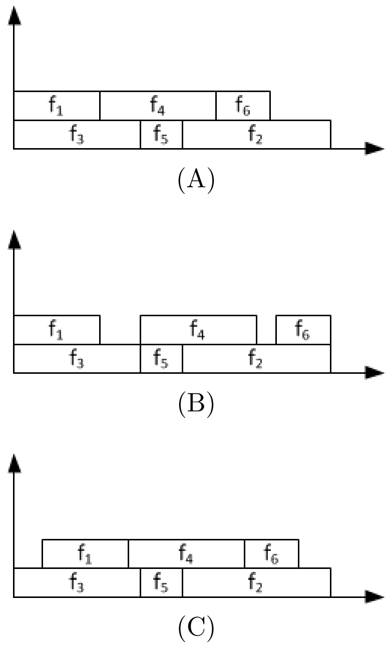

Figure 1: Feasible and infeasible solutions to the CAP

one row rather than two rows as in CAP, with a similar objective; and the double row facility layout problem (DRFLP) which is very similar to the CAP except that conditions (i) and (ii) above are not imposed on the layout. Thus all three layouts (A), (B), and (C) are feasible layouts in a DRFLP, although not for the CAP.

The CAP has recently been introduced in the literature, and we are aware of only one paper (Amaral 2012) dealing with the problem. The paper introduces the CAP, and provides a mixed integer programming formulation for it. It also presents two neighborhood search based heuristics *Heuristic_1* and *Heuristic_2* to arrive at solutions to CAP instances which are too large to be solved by exact approaches. Amaral (2012) uses problem instances commonly used as benchmark instances for the SRFLP to perform computational experiments. Using cplex 12.1.0 the integer programming formulation solves all benchmark instances with up to 13 facilities to optimality. The paper also reports heuristic solutions to benchmark instances with 30 facilities. The related SRFLP has been widely studied in the literature (see Kothari and Ghosh 2012a, for a detailed review of the SRFLP literature). At present, the studies of Amaral and Letchford (2012) and Hungerländer and Rendl (2011) provide the best results among exact algorithms for the SRFLP, and the genetic algorithm of Ozcelik (2011) and scatter search algorithms of Kothari and Ghosh (2012b) provide the best results among heuristic approaches. The DRFLP was introduced in Chung and Tanchoco (2010). The paper presents a mixed integer programming formulation for the problem, which was subsequently corrected in Zhang and Murray (2012). Chung and Tanchoco (2010) solved their mixed integer program using cplex 10.2, and also developed five heuristic approaches to solve the DRFLP. Amaral (2011) proposed an alternate mixed integer program, and demonstrated that it was superior to the formulation in Chung and Tanchoco (2010).

Since the CAP is NP-hard, and since Amaral (2012) reports that cplex 12.1.0 could not obtain an optimal solution for a CAP instance with 15 facilities even after 8.6 hrs of execution time, it is useful to consider heuristics to solve medium to large size CAP instances. We propose two metaheuristics to solve the CAP. The performance of various metaheuristics for the related SRFLP shows that population based metaheuristics and hybrid metaheuristics perform better than other metaheuristics (see, e.g. Ozcelik 2011, Kothari and Ghosh 2012b). So we develop a hybrid genetic algorithm and a scatter search algorithm with path-relinking for the CAP in this paper. In the next section we describe these two algorithms for the CAP. In Section 3 we present results of our computational experiments on benchmark instances. We then conclude the paper in Section 4 with a summary of our work and future research directions.

2 Our algorithms for the CAP

In this section we describe two population based metaheuristics for the CAP. The first is a genetic algorithm which uses basic operators of selection, crossover, and mutation, and has a local search component built into it. The second algorithm that we describe here is a scatter search algorithm. In each iteration of scatter search, the solutions in the reference set are subjected to a path-relinking operation in order to improve their quality. Thus both the algorithms that we describe here are hybrid algorithms. In the remainder of the section, we first describe our solution representation, and then provide a detailed description of the algorithms.

Solution representation

Both algorithms require us to represent solutions of the CAP in a manner in which the algorithms can manipulate. We describe each solution as a permutation of the n elements in F. A representation $S = (f_1, f_2, f_3, \ldots, f_{n-1}, f_n)$ represents a solution in which one of the rows has facilities $f_1, f_2, \ldots, f_k$ in that order from left to right, and the other row has facilities $f_n, f_{(n-1)}, \ldots, f_{(k+1)}$ in that order from left to right. This solution is shown pictorially in Figure 2. One apparent shortcoming of this

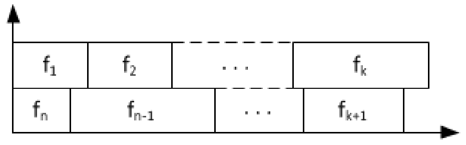

Figure 2: The solution represented by $(f_1, f_2, f_3, \ldots, f_{n-1}, f_n)$

representation is that each permutation actually represents $(n+1)$ solutions to the CAP, depending on the value of k. However, all but one of the solutions can never be an optimal solution to the CAP. Consider for example, all solutions in which one row contains facilities $f_1, f_2, \ldots, f_{(k-1)}, f_k$ in that order from left to right, and the other row contain facilities $f_n, f_{(n-1)}, \ldots, f_{(k+1)}$ in that order from left to right. If $\sum_{i=1}^{k-1} l_i > \sum_{i=k+1}^{n} l_i$, this solution is inferior to the solution formed by shifting the facility f_k from its current position to the position to the right of facility $f_{(k+1)}$. Using a similar argument, all solutions of the same form, in which $\sum_{i=1}^{k} l_i < \sum_{i=k+2}^{n} l_i$ are sub-optimal. This leaves exactly one solution from each permutation which is definitely a candidate optimal solution. This solution is constructed from the permutation by the OBTAIN-CAP-SOLUTION algorithm as follows. Initially, the first facility in the permutation is assigned to one row (called the top row) and the the last facility is assigned to the other (called the bottom row). The solution is then constructed in $n-2$ steps. At the beginning of an iteration, if the sum of lengths of the facilities in the top row is larger than the sum of the lengths of the facilities in the bottom row, the last unassigned facility in the permutation is added at the end of the bottom row; otherwise the first unassigned facility in the permutation is added to the end of the top row. We provide a pseudocode for the OBTAIN-CAP-SOLUTION algorithm below. In the pseudocode, $f_{[j]}$ denotes the facility in the j-th position in the permutation. T is an ordered list of facilities in the top row, B is an ordered list of facilities in the bottom row, and L_T and L_B denote the sums of the lengths of the facilities in the top row and bottom row respectively. The operation of adding a facility f to the end of row R is denoted by $R \oplus \{f\}$ in the pseudocode.

ALGORITHM OBTAIN-CAP-SOLUTION

Input: A CAP instance (F, L, C); a permutation of the facilities in F.
Output: A solution to the CAP corresponding to the input permutation.

Steps:

```
1. begin
2.     set $T \leftarrow (f_{[1]})$; $B \leftarrow (f_{[n]})$; $L_T \leftarrow l_{[1]}$; $L_B \leftarrow l_{[n]}$; $u_T \leftarrow 2$; $u_B \leftarrow n-1$;
3.     for ($iter$ from 1 to $n-2$) do begin
4.         if ($L_T > L_B$) then
5.             $B \oplus \{f_{[u_B]}\}$; $L_B \leftarrow L_B + l_{[u_B]}$; $u_B \leftarrow u_B - 1$;
6.         else
7.             $T \oplus \{f_{[u_T]}\}$; $L_T \leftarrow L_T + l_{[u_T]}$; $u_T \leftarrow u_T + 1$;
8.     end;
9. end.
```

Given this solution representation, we are now in a position to describe the genetic algorithm and the scatter search algorithm that we propose in this work.

CAP-GA: A genetic algorithm incorporating local search

A genetic algorithm (see, e.g., Goldberg (1989)) is an evolutionary search algorithm which simulates natural evolution based on the principle of the survival of the fittest. Solutions to the problem are coded as individuals, and a population of individuals is iteratively improved using genetic operators such as selection, crossover, and mutation, to create individuals of high fitness. At the end of the genetic algorithm run, the fittest individual observed during the run is output by the algorithm.

We present a genetic algorithm called CAP-GA to solve the CAP. In our algorithm, individuals are permutations of the facilities in the CAP instance. The solution corresponding to each individual is obtained using the OBTAIN-CAP-SOLUTION presented above. Since the CAP is a minimization problem, lower cost solutions have higher fitness values. The fitness of an individual is taken as the reciprocal of the cost of the solution corresponding to that individual.

CAP-GA is a generic genetic algorithm with the exception that periodically, all the individuals in the generation are subjected to local search to improve their fitnesses. The pseudocode of the algorithm is provided below, and details of several of the steps are explained thereafter.

ALGORITHM CAP-GA

Input: A CAP instance (F, L, C); population size N; maximum number of generations $maxgen$; period k.

Output: An approximation of an optimal solution to the CAP instance.

Steps:

```
1. begin
2.     create the initial population $init\_pop$ of $N$ individuals by generating $N$ permutations
       of facilities in $F$;
3.     set the current population $pop \leftarrow init\_pop$;
```

```
4.      for ($iter$ from 1 to $maxgen$) begin
5.          create a mating pool of $N$ individuals from $pop$;
6.          perform crossover to generate a child pool of $N$ individuals;
7.          perform mutation on select individuals in the child pool;
8.          merge the mating pool and child pool, and select $N$ individuals with lowest cost
            from the merged pool to replace all the individuals in $pop$;
9.          if ($iter$ is divisible by $k$) then improve all the individuals in $pop$ using a local
            improvement operation;
10.     end;
11.     output the solution corresponding to the fittest individual encountered in the process
        and terminate.
12. end.
```

Step 1 of the algorithm CAP-GA creates an initial population of N randomly generated individuals. This generation is assigned to the current population, which is improved by the code in Steps 5 through 9 until a stopping condition is reached.

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. This is done in Step 5 of the algorithm. In CAP-GA, the mating pool is created using a binary tournament selection (see, e.g. Deb 2001) in which the fitter of the two randomly selected individuals from the population enters the mating pool. This may allow multiple copies of the same solution in the mating pool. The solutions in the mating pool thus generated are subsequently subjected to crossover and mutation operations in Steps 6 and 7 to create candidate solutions for the next generation. We use the partially matched crossover (PMX) operator (Larrañaga et al. 1999) in Step 6 to cross individuals with a user-defined crossover probability p_c to create a child pool. After generating the child pool, each child is subjected to a mutation operator in Step 7 of CAP-GA. With a user-defined mutation probability p_m, a subset of facilities in the solution are selected and they are then randomly rearranged at those selected locations to yield a new solution. This mutation is known in the literature as scramble mutation (Larrañaga et al. 1999). Step 8 of CAP-GA is an elite preservation operation, which ensures that individuals with high fitness in the previous generation are carried over to the next generation, and individuals with low fitness obtained after crossover and mutation are eliminated. This ensures that the average fitness of individuals in populations does not degrade over generations. Step 9 is a local improvement operation that is carried out on every individual of every k-th generation. This is done to improve the fitnesses of individuals in every population. We use an insertion neighborhood for local improvement in this step since experience in the literature suggests that the insertion neighborhood is the best neighborhood structure for the related SRFLP (Kothari and Ghosh 2012c). The stopping condition that we use in CAP-GA is a maximum number of iterations, so that the algorithm terminates after generating *maxgen* generations.

CAP-SS: A scatter search algorithm with path relinking

Scatter search is an evolutionary technique which orients its exploration relative to a set of permutations, called the reference set. The reference set typically consists of good solutions obtained by other methods. The criterion of "good" may refer not only to the cost of the permutation, but also to certain other properties that the solution or a set of solutions may possess. Most scatter search algorithms for combinatorial optimization problems use the scatter search template by Glover (1998) as a reference for building the algorithm. Our CAP-SS algorithm uses the same template. It also applies path relinking (Glover 1977) to each reference set generated by the algorithm to explore those regions of the search space which have not been explored by scatter search. We provide a pseudocode of CAP-SS below, and explain the important components of the algorithm thereafter.

ALGORITHM CAP-SS

Input: A CAP instance (F, L, C); initial population size P_Size; reference set size B_Size.

Output: The best permutation which has the lowest cost among all the permutations encountered in the algorithm.

Steps:

```
1.  begin
2.      generate an initial population with $P\_Size$ permutations using a diversification
        generation method;
3.      improve each permutation using the local improvement operation;
4.      construct the reference set RefSet of size $B\_Size$ from the initial population;
5.      set flag ←TRUE;
6.      while (flag = TRUE) do
7.          create all subsets of size 2 using the permutations in the RefSet;
8.          for ($i$ from 1 to $\binom{B\_Size}{2}$) do begin
9.              flag ←TRUE;
10.             use the PMX crossover method as a subset combination method to generate a
                permutation from the permutations in the $i$-th subset;
11.             use the improvement method to obtain a local optimum $\Pi$ from the
                permutation generated in the previous step;
12.             add $\Pi$ to the reference set;
13.             remove the highest cost solution $\Pi_r$ from the reference set;
14.             if (cost($\Pi$) < cost($\Pi_r$)) then set flag ←FALSE;
15.         end;
16.         improve the quality of solutions in RefSet through path relinking;
17.     end while;
18.     output the solution corresponding to the lowest cost permutation in RefSet;
19. end.
```

Step 2 of CAP-SS generates an initial population of permutations using a diversification generation method. This determines the quality of permutations in the reference set *RefSet* and ensures diversification in the search process. We use the concept of deviation distances (see Sörensen (2007)) to measure the diversification in a set of permutations. We define the distance between two permutations Π_1 and Π_2 as the minimum of the deviation distance between Π_1 and Π_2 and the deviation distance between Π_1 and the permutation obtained by reversing Π_2. Using this distance measure and two user-specified numbers U_Size and E_Size, we generate a diversified set of good permutations which serves as the initial population for our scatter search algorithm. Starting from the permutation $\Pi = \{f_1, f_2, \ldots, f_n\}$, we generate U_Size permutations by randomly interchanging facilities in Π. We then choose E_Size lowest cost permutations among them to form a set of elite permutations. We finally generate the initial population of size P_Size by choosing permutations from the elite set in a way such that the minimum distance between any two permutations in the population is as high as possible.

In Step 3, and later in Step 11, CAP-SS improves solutions using local improvement. As in the case of CAP-GA, the local improvement method is simply a local search using the insertion neighborhood structure.

In Step 4, CAP-SS creates a reference set *RefSet* from the population generated in Step 2. *RefSet* contains B_Size permutations including the best and the most diverse among the solutions in the initial population. To do this it accepts two user specified parameters B_1 and B_2 such that $B_1 + B_2 = B\_Size$. The improved permutations in the initial population are sorted in non-decreasing order of their costs and the first B_1 and last B_2 permutations in the sorted list are selected to be the members of *RefSet*.

In Step 7, the subset generation method generates subsets of *RefSet* of cardinality 2. Given a subset of the *RefSet*, a new solution is constructed in Step 10 using the PMX crossover operation as the solution combination method. The permutation thus obtained is improved using local search in Step 11. In Steps 12 and 13, CAP-SS updates *RefSet*. To do this it adds the new permutation in *RefSet* and removes a worst permutation from *RefSet*. It then checks in Step 14 whether the updating of *RefSet* improves the average cost of permutations in *RefSet*. Once CAP-SS processes all the $\binom{B\_Size}{2}$ subsets of *RefSet* in this manner, it subjects the permutations in *RefSet* to path relinking in Step 16. It also checks whether there was any improvement in the average cost of permutations in *RefSet* due to the processing. If so, then the *RefSet* is updated and path relinking operation is repeated. Otherwise, CAP-SS outputs the solution corresponding to the best permutation in the latest *RefSet*.

Path relinking is a path construction technique which aims to generate additional solutions which may not have been considered by the algorithm that generated the initial pool of solutions. It typically chooses two solutions, marks one as the initiating solution and the other as the guiding solution, and generates a path from the initiating solution to the guiding solution through a set of moves. A move in the path starts with a solution and generates another solution in which some attribute of the initiating solution is replaced with some attribute of the guiding solution. The idea is to introduce attributes of the guiding solution in each move to finally reach the guiding solution from the initiating solution or to extend beyond it. For path relinking in CAP-SS, the initial pool of solutions is *RefSet*. The permutations in *RefSet* are sorted in non-decreasing order of their costs. Every solution in the first half of the sorted list is used once as an initiating solution. For every initiating solution, the guiding solution is that solution in the second half of the list which has the maximum deviation distance (see Sörensen (2007)) from the initiating solution. A path between the initiating solution and the guiding solution is obtained by executing moves that are necessary to transform the initiating solution into guiding solution. For example, if in an intermediate solution in the path, a facility f_i is in the i-th position in the permutation, and in the guiding solution f_i is in the j-th position in the permutation, then a move will interchange f_i with the facility in the j-th position in the initiating permutation to obtain an intermediate solution.

In the next section, we report the results of our computational experiments with CAP-GA and CAP-SS on benchmark instances.

3 Computational experience

Several parameter values need to be specified in the pseudocodes presented in the previous section to yield implementations. In our CAP-GA implementation, we set the population size N to 60. The crossover probability p_c was chosen at random from the set $\{0.65, 0.75, 0.85, 0.95\}$ and the probability of mutation p_m was chosen randomly from the interval $(0.00, 0.05]$. The parameter k was set to 5 and the maximum number of generations *maxgen* to 500. We report the best results obtained after 200 runs of CAP-GA on each problem instance. For CAP-SS the size of the initial population, P_Size, was set to 100 and the size of *RefSet* B_Size to 50. In order to generate the initial population, the values of the parameters *U_Size* and *E_Size* were taken as 1000 and 500 respectively. In order to generate the *RefSet*, the values of parameters B_1 and B_2 were taken as 25 each.

We coded CAP-GA and CAP-SS in C, and compiled them using the gcc 4.5.1 compiler. We then ran these codes on a machine with an Intel Core i5 2400 processor at 3.1 GHz running Window 7. The performances of CAP-GA and CAP-SS are measured on two factors, the cost of the solution they output, and the execution times they require.

We use data from SRFLP benchmark instances to test the performance of CAP-GA and CAP-SS. This is because, the CAP and the SRFLP require the same data, and hence benchmark SRFLP instances can be used as benchmarks to test performance of algorithms for the CAP. Amaral (2012) for example, uses data from SRFLP benchmark instances to test the performance of algorithms in his paper. There are many sets of benchmark instances for the SRFLP, and in this paper, we test our algorithms on SRFLP benchmark instances with less than 50 facilities. In the remainder of this section, we report our computational experience on different sets of benchmark instances.

3.1 Instances solved in Amaral (2012)

Amaral (2012) uses four instances described in Simmons (1969) and generates four other instances of sizes 12 and 13 to test the performance of the mixed integer formulation proposed therein. The mixed integer programming approach obtained optimal solutions for all these instances. Our CAP-GA and CAP-SS implementations also produced optimal solutions to these instances, though being heuristics, they required much less time than those reported in Amaral (2012). The mixed integer programming approach in Amaral (2012) failed to obtain an optimal solution to an instance of size 15. Using heuristic approaches, the paper reported an upper bound to the cost of an optimal solution which was better than that output by cplex 12.1.0 after 8.6 hrs of execution. Amaral (2012) also experiments with five instances of size 30 reported in Anjos and Vannelli (2008). In Table 1, we present the costs of the best solutions reported in Amaral (2012), and those obtained CAP-GA and CAP-SS. We also report the execution times in cpu seconds required for all these instances.

Table 1: Results for instances experiemented with in Amaral (2012)

			Amaral (2012)		CAP-GA		CAP-SS	
Instance	Source	Size	cost	time	cost	time	cost	time
S9	Simmons (1969)	9	1181.5*	18.42	1181.5	17.61	1181.5	0.19
S9H	Simmons (1969)	9	2294.5*	1145.94	2294.5	17.54	2294.5	0.24
S10	Simmons (1969)	10	1374.5*	62.75	1374.5	21.96	1374.5	0.27
S11	Simmons (1969)	11	3439.5*	496.22	3439.5	29.21	3439.5	0.53
Am12a	Amaral (2012)	12	1529.0*	1869.31	1529.0	40.20	1529.0	0.90
Am12b	Amaral (2012)	12	1609.5*	1412.06	1609.5	38.30	1609.5	0.80
Am13a	Amaral (2012)	13	2467.5*	10298.01	2467.5	50.30	2467.5	1.50
Am13b	Amaral (2012)	13	2870.0*	5144.13	2870.0	53.76	2870.0	1.35
P15	Amaral (2008)	15	3195.0†	2.41	3195.0	77.99	3195.0	2.18
N-30-01	Anjos and Vannelli (2008)	30	4115.0‡	6070.50	4115.0	1506.69	4115.0	118.68
N-30-02	Anjos and Vannelli (2008)	30	10779.5‡	7412.60	10779.5	1565.07	10779.5	150.06
N-30-03	Anjos and Vannelli (2008)	30	22702.0‡	7902.40	22702.0	1611.19	22702.0	163.10
N-30-04	Anjos and Vannelli (2008)	30	28401.5‡	8270.70	28401.5	1628.00	28401.5	189.20
N-30-05	Anjos and Vannelli (2008)	30	57400.0‡	8562.30	57400.0	1672.68	57400.0	188.97

* indicates the cost of an optimal solution.
† indicates the cost of the solution obtained using *Heuristic_1* with 2-opt.
‡ indicates the cost of the solution obtained using *Heuristic_2* with 2-opt.

We see from Table 1 that both CAP-GA and CAP-SS match the costs of the best solutions reported in Amaral (2012) for all the instances. For the Simmons instances and the four Amaral instances, the time required by the two heuristics is understandably lower than the times required to solve the mixed integer programming formulations. For these instances, the times required by CAP-SS is much lower than those required by CAP-GA. They are also lower than the times reported for *Heuristic_2* although not for *Heuristic_1*. For the problem instance with size $n = 15$, CAP-SS required less time than both *Heuristic_1* and *Heuristic_2*, although CAP-GA required more time. For the instances of size $n = 30$, the execution times required by both CAP-GA and CAP-SS are much lower than those reported in Amaral (2012) both for *Heuristic_1* and for *Heuristic_2*. For these problems, CAP-SS required significantly lower execution times than CAP-GA. Hence for problems discussed in Amaral (2012), CAP-SS is clearly the superior implementation.

3.2 Other benchmark instances with sizes $25 \leq n < 50$

There are several sets of medium size SRFLP benchmark instances apart from the instances of size 30 experimented with in Amaral (2012). In this section, we report the performance of CAP-GA and CAP-SS on these instances. The instances that we consider are (a) a set of five instances each of size 25 reported in Anjos and Vannelli (2008); (b) two sets of three instances each of sizes 33 and 35 reported in Amaral (2008); (c) a set of five `ste` instances each of size 36 reported in Anjos and Yen (2009); (d) a set of five instances each of size 40 reported in Hungerländer and Rendl (2011); and (e) two sets of five `sko` instances each of sizes 42 and 49 reported in Anjos and Yen (2009). We report our computational experience with CAP-GA and CAP-SS with these instances in Table 2.

The results in Table 2 bring out interesting properties of the CAP-GA and CAP-SS implementations. The costs output by both the implementations are identical for the instances with size 25. Thereafter, CAP-SS outputs better solutions for most of the instances. Of the 26 instances with sizes $n > 25$, CAP-SS produces better solutions than CAP-GA in 19 instances and matches the cost of CAP-GA output in another four instances. It produces worse solutions in the remaining three instances. The execution times required by CAP-SS is lower than those required by CAP-GA. This shows that CAP-SS is superior to CAP-GA for medium size CAP instances.

A closer look at the algorithms gives a different insight. The output of CAP-GA is obtained by running the genetic algorithm 200 times. So the average cost of a genetic algorithm run incorporating local search for the CAP is 1/200-th the time reported in the tables. The output of CAP-SS is from a single scatter search run incorporating path relinking. So the CAP-GA algorithm requires less time per run than the CAP-SS algorithm. We feel that the difference is the execution times for CAP-GA and CAP-SS applied to the same instance is mainly due to the number of local search operations carried out by both the implementations. Local search for the CAP is computationally expensive, with an order of complexity of $O(n^4)$ where n denotes the size of the instance. The average number of local search operations per generation for CAP-GA is $O(N)$ while that for CAP-SS is $O(B\_Size^2)$, which is much higher. If we compare the ratios of times taken for a single run by the two algorithms, we see that, on average, the ratio of the time required for one run of CAP-SS to that required by one run of CAP-GA is 15.7 for instances of size 25 and 40.2 for instances of size 49. The costs of the solutions output by CAP-SS for these instances are better than those output by CAP-GA, but the difference in the costs is on average less than 0.2% of the costs of the solutions output, and do not increase appreciably with an increase in the size of the instance. Thus, we conjecture that CAP-GA will be preferred over CAP-SS for large instances.

4 Summary

In this paper, we have studied metaheuristics for the corridor allocation problem. The problem is to arrange a set of facilities in two rows along a straight corridor such that a weighted sum of the

Table 2: Results for benchmark instances with sizes $n \leq 25 < 50$

			CAP-GA		CAP-SS	
Instance	Source	Size	cost	time	cost	time
N-25-01	Anjos and Vannelli (2008)	25	2302.0	618.58	2302.0	41.50
N-25-02	Anjos and Vannelli (2008)	25	18595.5	736.25	18595.5	59.91
N-25-03	Anjos and Vannelli (2008)	25	12114.0	677.39	12114.0	63.42
N-25-04	Anjos and Vannelli (2008)	25	24192.5	742.12	24192.5	56.20
N-25-05	Anjos and Vannelli (2008)	25	7819.0	719.97	7819.0	53.14
P1	Amaral (2008)	33	30282.5	2287.84	30282.5	285.32
P2	Amaral (2008)	33	33974.0	2349.55	33978.0	221.66
P3	Amaral (2008)	33	35065.5	2330.13	35060.5	322.39
P4	Amaral (2008)	35	34671.5	2967.47	34673.5	478.29
P5	Amaral (2008)	35	30803.0	2801.70	30781.0	390.76
P6	Amaral (2008)	35	34425.5	3010.98	34424.5	446.38
ste-36-01	Anjos and Yen (2009)	36	4966.0	3475.61	4966.0	423.74
ste-36-02	Anjos and Yen (2009)	36	88262.0	3237.71	87489.0	721.35
ste-36-03	Anjos and Yen (2009)	36	50696.5	3279.40	50127.5	674.71
ste-36-04	Anjos and Yen (2009)	36	46890.5	3446.11	46824.5	504.18
ste-36-05	Anjos and Yen (2009)	36	44783.5	3407.52	44488.5	593.57
N-40-01	Hungerländer and Rendl (2011)	40	53763.5	5597.52	53722.5	782.26
N-40-02	Hungerländer and Rendl (2011)	40	48916.0	5448.85	48908.0	773.86
N-40-03	Hungerländer and Rendl (2011)	40	39259.5	5330.11	39255.5	760.12
N-40-04	Hungerländer and Rendl (2011)	40	38355.0	5459.45	38354.0	741.42
N-40-05	Hungerländer and Rendl (2011)	40	51523.0	5585.37	51507.0	836.69
sko-42-01	Anjos and Yen (2009)	42	12731.0	7174.64	12731.0	1249.04
sko-42-02	Anjos and Yen (2009)	42	108049.5	6815.77	108020.5	1533.29
sko-42-03	Anjos and Yen (2009)	42	86667.5	6854.25	86667.5	956.28
sko-42-04	Anjos and Yen (2009)	42	68769.0	6509.35	68733.0	1141.34
sko-42-05	Anjos and Yen (2009)	42	124099.5	7052.59	124058.5	1236.47
sko-49-01	Anjos and Yen (2009)	49	20472.0	14934.38	20479.0	2683.93
sko-49-02	Anjos and Yen (2009)	49	208270.0	14338.86	208081.0	2733.87
sko-49-03	Anjos and Yen (2009)	49	162267.0	14351.91	162196.0	2877.19
sko-49-04	Anjos and Yen (2009)	49	118311.5	14290.03	118264.5	3716.16
sko-49-05	Anjos and Yen (2009)	49	332990.0	14913.18	332855.0	2596.39

distance between each pair of facilities is minimized. We have presented two hybrid metaheuristics, a genetic algorithm with local search, and a scatter search algorithm with path relinking to solve the problem.

We have performed computational experiments with both algorithms on several sets of benchmark instances, including all the instances that were used in Amaral (2012). Our experience suggests that for small and medium size instances (with less than 50 facilities), scatter search with path relinking is the heuristic of choice. We conjecture that for larger instances genetic algorithms with local search may prove to be more computationally tractable.

There are several directions in which the literature on corridor allocation problems can be extended. First, the conjecture that we have made in the previous paragraph can be tested. Second, metaheuristics based on variable neighborhood search can be looked into. Recently, a book-keeping technique described in Kothari and Ghosh (2012c) for the related single row facility layout problem has helped to reduce the order of complexity of performing neighborhood search from $O(n^4)$ to $O(n^3)$ for 2-opt and insertion neighborhoods. Such a technique would prove useful in implementing neighborhood search for the corridor allocation problem.

References

Amaral, A. (2011). Optimal solutions for the double row layout problem. *Optimization Letters*, pages 1–7.

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. (2012). The corridor allocation problem. *Computers & Operations Research*. 10.1016/j.cor.2012.04.016.

Amaral, A. R. S. and Letchford, A. N. (2012). A polyhedral approach to the single row facility layout problem. *Mathematical Programming*. Available at http://dx.doi.org/10.1007/s10107-012-0533-z.

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.

Chung, J. and Tanchoco, J. (2010). The double row layout problem. *International Journal of Production Research*, 48(3):709–727.

Deb, K. (2001). *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons Ltd, Chichester, England.

Glover, F. (1977). Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8(1):156–166.

Glover, F. (1998). A template for scatter search and path relinking. *Lecture notes in computer science*, 1363:13–54.

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

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

Kothari, R. and Ghosh, D. (2012a). Insertion based Lin-Kernighan heuristic for single row facility layout. *Computers & Operations Research*. 10.1016/j.cor.2012.05.017.

Kothari, R. and Ghosh, D. (2012b). Scatter search algorithms for the single row facility layout problem (w.p. no. 2012-04-01). Ahmedabad, India: IIM Ahmedabad, Production & Quantitative Methods. Available at `www.optimization-online.org/DB_HTML/2012/03/3400.html`.

Kothari, R. and Ghosh, D. (2012c). Tabu search for the single row facility layout problem using exhaustive 2-opt and insertion neighborhoods. *European Journal of Operational Research*. 10.1016/j.ejor.2012.07.037.

Larrañaga, P., Kuijpers, C. M. H., Murga, R. H., Inza, I., and Dizdarevic, S. (1999). Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170.

Ozcelik, F. (2011). A hybrid genetic algorithm for the single row layout problem. *International Journal of Production Research*.

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

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.

Zhang, Z. and Murray, C. (2012). A corrected formulation for the double row layout problem. *International Journal of Production Research*, 50(15):4220–4223.