RAIRO Operations Research
RAIRO Oper. Res. **35** (2001) 127-142

SOLVING THE SIMPLE PLANT LOCATION PROBLEM BY GENETIC ALGORITHM

Jozef Kratica[1], Dušan Tošic[2], Vladimir Filipović[2] and Ivana Ljubić[3]

Communicated by P. Tolla

Abstract. The simple plant location problem (SPLP) is considered and a genetic algorithm is proposed to solve this problem. By using the developed algorithm it is possible to solve SPLP with more than 1000 facility sites and customers. Computational results are presented and compared to dual based algorithms.

Keywords: Simple plant location problem, genetic algorithms, combinatorial optimization.

Mathematics Subject Classification. 90B80, 90C27, 90C59.

1. Introduction

1.1. Problem formulation

Consider a set $I = \{1, \ldots, m\}$ of candidate sites for facility location, and a set $J = \{1, \ldots, n\}$ of customers. Each facility $i \in I$ has a fixed cost f_i. Every customer $j \in J$ has a demand b_j, and c_{ij} is the unit transportation cost from facility i to

Received February, 2000. Accepted February, 2001.

[1] University of Belgrade, Serbian Academy of Sciences and Arts, Institute of Mathematics, Strumicka 92/5, 11 000 Belgrade, Yugoslavia; e-mail: `jkratica@matf.bg.ac.yu`

[2] University of Belgrade, Faculty of Mathematics, Studentski trg 16, 11 000 Belgrade, Yugoslavia; e-mail: `{dtosic|vladaf}@matf.bg.ac.yu`

[3] Institute for Computer Graphics, Faviritensstasse 9, Vienna, Austria;
e-mail: `ivana@ads.tuwien.ac.at`

customer j. Without a loss of generality we can normalize the customer demands to $b_j = 1$ [6].

It has to be decided:

– facilities to be established and
– quantities to be supplied from facility i to customer j

such that the total cost (including fixed and variable costs) is minimized.

Mathematically, the SPLP is formulated on the following way:

$$\min\left(\sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} + \sum_{i=1}^{m} f_i y_i\right) \tag{1}$$

subject to:

$$\sum_{i=1}^{m} x_{ij} = 1, \qquad \text{for every } j \in J; \tag{2}$$

$$0 \le x_{ij} \le y_i \text{ and } y_i \in \{0,1\}, \quad \text{for every } i \in I \text{ and every } j \in J; \tag{3}$$

where:

x_{ij} represents the quantity supplied from facility i to customer j;

y_i indicates whether facility i is established ($y_i = 1$) or not ($y_i = 0$).

Let the set of established facilities be $E = \{i | y_i = 1\}$ with cardinality $e = |E|$.

Although some special cases of the SPLP are solvable in polynomial time (see [3, 11, 17, 26]) in general, it is a NP-hard problem [22]. Genetic algorithms are used to solve some NP-hard problems in [10, 20]. The most recent researches show that combination of GA and local search can be successful in solving NP-hard problems (see [12, 15, 24, 25]).

1.2. Genetic algorithms

Genetic Algorithms (GAs) are robust and adaptive methods that can be used to solve search and optimization problems. GAs work with a population of individuals (usually 10–200), each representing a problem's possible solution. Each individual has an assigned fitness value according to the quality of the solution. The population evolves towards better solutions by means of randomized processes of selection, crossover, and mutation.

During the forming of a new population, the selection mechanism favors better fitted individuals to reproduce more often than the worse ones. Crossover allows the mixing of parental information when it is passed to their descendants. The result of crossover is a randomized exchange of genetic material between individuals with the possibility that good solutions can generate even better ones.

Mutation modifies individual's genetic material with some small probability p_{mut}. Its role is to restore lost or unexplored genetic material into the population. Mutation should prevent the premature convergence of the GA to suboptimal

```
Input_Data();
Population_Init();
**while not** Finish() **do**
/* In the following sentence $N_{\text{pop}}$ is the number of individuals
   in a population and $p_i$ is fitness of $i$th individuals.        */
        **for** $i := 1$ **to** $N_{\text{pop}}$ **do** $p_i$:=Objective_Function(i) **endfor**;
        Fitness_Function();
        Selection();
        Crossover();
        Mutation();
**endwhile**
Output_Data();
```

FIGURE 1. Simple form of GA.

solutions. The first generation is usually randomly initialized. For detailed informations about GA see [4].

Informal description leads to the rough outline of a GA given in Figure 1.

1.3. SPLP literature survey

The SPLP is a well-known combinatorial optimization problem, and many approaches have been proposed for solving it. The problem is also known as uncapacitated warehouse location problem or uncapacitated facility location problem. The presentation of all important contributions relevant to SPLP lies beyond the scope of this paper. Some important survey articles are [1, 8, 9, 14, 16, 22]. We are going to mention only several efficient and well-known methods to solve SPLP.

The DUALOC algorithm by Erlenkotter [13] has been the fastest algorithm for solving SPLP for a long time. This algorithm is based on a linear programming dual formation (LP dual) in condensed form that evolved in simple ascent and adjustment procedure. If ascent and adjustment procedures do not find optimal solution, Branch-and-Bound (BnB) procedure completes the solution process.

Guignard [18] proposed to strengthen the separable Lagrangean relaxation of the SPLP by using Benders inequalities generated during a Lagrangean dual ascent procedure. The coupling that technique with a good primal heuristic could reduce integrality gap.

In [27] Simao and Thizy presented streamlined dual simplex algorithm designed on the basis of a covering formulation of the SPLP. Their computational experience with standard data sets indicates the superiority of dual approaches.

Körkel [21] showed how to modify a primal-dual version of Erlenkotter's exact algorithm to get an improved procedure. The computational experience with large-scale problem instances indicated that speedup to DUALOC is significant (more than one order of magnitude).

Coon and Cornuejols present a method based upon the exact solution of the condensed dual of LP relaxation via orthogonal projections in [7].

Alves and Almeida in [2] have compared the results of four versions of their simulated annealing algorithm with those of some well-known heuristic methods. The quality of the simulated annealing solutions is very good, but the computing time is significantly longer in comparison to the other ones.

In [19] Holmberg used a primal-dual solution approach based on decomposition principles. He fixed some variables in the primal subproblem and relaxed some constraints in the dual subproblem. By fixing their Lagrange multipliers, both of these problems become easier to solve than the original one. The computational tests showed advantageous in comparison to the dual ascent method of Erlenkotter.

2. GA implementation

Outline of GA implementation for solving SPLP is schematically described in Figure 2.

```
Input_GA_Parameters();
Input_SPLP_Data();
Population_Init();
while not Finish() do
      for $i := N_{\text{elite}}+1$ to $N_{\text{pop}}$ do
/*    $N_{\text{elite}}$ - number of elite individuals and
      string(i) - genetic code of $i$-th individual      */
           if Contain(cache_memory, string(i)) then
              $p_i$:= Get(cache_memory, string(i));
           else
              $p_i$:= Objective_Function(i);
              Put(cache_memory, string(i));
           endif
      endfor
      Fitness_Function_and_Rank_Based_Selection();
      Uniform_Crossover();
      Simple_Mutation();
endwhile
Output_Data();
```

Figure 2. Outline of GA implementation.

2.1. Representation

The binary encoding of facility sites is used for the given problem (SPLP). Each individual is represented by the binary string, where 1 denotes that particular facility is established, while 0 shows it is not. For the faster execution an individual string is allocated in 32-bit words.

The array y_i ($i = 1, 2, \dots, m$) is obtained from the individual string. This array indicates the established facilities. Since the capacity of the facilities has no limit, if every customer chooses the most suitable facility (with minimal transportation cost), the total cost is minimal. Note that the procedure for choosing of the most suitable facility participated only established facilities ($y_i = 1$). Therefore, the genetic code contains only y_i and the value of x_{ij} is calculated during the evaluation of objective value function.

2.2. Objective value function

The choice of the data structure is very important for the fast implementation of objective value function. Beside transportation cost matrix this implementation also uses the indexed lists of facility sites. These lists are sorted in nondecreasing order according to transportation cost for every customer.

In the program's initialization part, built-in function *qsort()* produce indexed lists from transportation cost matrix. The indexed lists require about 50% additional memory, but the computation of the objective value function is several times faster.

Two different methods for computing objective value have been proposed. The choice of particular method depends on the number of established facilities e. Threshold is $e_0 = c \cdot \sqrt{m}$, where c is constant from interval [0.4, 0.5]. Its value is obtained experimentally to achieve the best performance.

If e is large ($e > e_0$), algorithm is looking for the first facility in indexed list where is $y_i = 1$. Founded established facility i has minimal transportation cost for given customer j ($x_{ij} = 1$). The choosing procedure is fast because the array y is dense and only a few steps in searching the indexed list are needed. The number of steps for every customer is approximate m/e, and overall run-time complexity is $O\Big(m + n \cdot \frac{m}{e}\Big)$.

In the case of small number e, previous procedure gives poor results. It is slow because the array y is sparse and looking for adequate facility (with $y_i = 1$) in indexed list requires many steps. In that case another strategy is performed: instead of using indexed lists this procedure uses array of ordinals. The array O contains only established facilities ($O_k = i$ where y_i is k-th occurrence of 1 in the array y). Looking the most suitable facility for each customer is done by searching array of ordinals. If O_k is minimum in array O then $x_{ij} = 1$ for $i = O_k$. Array y is sparse, so the length of array O is small, which guaranties fast computation. The construction phase for array O has $O(m)$ time requirement. Nevertheless the program creates array O only once and uses it n times (once for every customer). This procedure requires e steps for each customer, so overall run-time complexity in this case is $O(m + n \cdot e)$.

It is easy to prove that total run-time complexity for objective value function, in the worst case, is $O(m + n \cdot \sqrt{m})$.

2.3. Selection and generation replacement strategy

Our implementation use rank-based selection as selection operator. This selection method produces better results on SPLP than other selection schemes. Especially large improvement is shown in comparison between experimental results for chosen selection and results for pure roulette selection. Those results concord with the direction in literature (see [4, 29]). The rank decreases linearly from the best individual $r_{\text{best}} = 2.5$ to the worst individual $r_{\text{worst}} = 0.712$ by step of 0.012. The individuals are chosen with chance proportional to its rank, as can be seen in Figure 3.

```
*QSort(population)*;
**for** *i:=1* **to** $N_{\text{pop}}$ **do**
    /* $f_i$ - fitness of the $i$-th individual */
    **if** *Duplicate(string(i))* **then** $f_i := 0$
    **else** $f_i :=$ *rank(i)*;
    **endif**
**endfor**
$\bar{f} :=$ *Sum_Fitnesses(population)* / $N_{\text{pop}}$;
**for** $i := 1$ **to** $N_{\text{elite}}$ **do**
    **if** $f_i \geq \bar{f}$ **then** $f_i := f_i - \bar{f}$
    **else** $f_i := 0$;
    **endif**
**endfor**
*Roulette_Selection_By_Rank(population)*;
```

Figure 3. Fitness function and rank-based selection.

The population size is $N_{\text{pop}} = 150$ individuals. GA is implemented with steady-state replacement of generations by using elitist strategy. In every generation only 1/3 of population (50 individuals) is replaced and 2/3 of population ($N_{\text{elite}} = 100$ individuals) remain from the previous generation. So, 50 worst ranked individuals in the population are replaced by the new ones. These new individuals (1/3 of population) are generated by means of the genetic operators crossover and mutation. Every elite individual is passed directly into the next generation, giving one copy of itself. To prevent undeserved domination of elite individuals over the population their fitness are decreased by formula (4):

$$f_i = \begin{cases} f_i - \bar{f}, \ f_i > \bar{f} \\ \\ 0, f_i \leq \bar{f} \end{cases} \qquad 1 \leq i \leq N_{\text{elite}} \tag{4}$$

where $\bar{f} = \frac{1}{N_{\text{pop}}} \sum_{i=1}^{N_{\text{pop}}} f_i$ is average fitness in entire population.

In this implementation duplicate individual strings are discarded, and more diversity of the population is maintained to avoid premature convergence. Particular individual is discarded by setting its fitness to zero. Therefore, the duplicate individual strings are not removed physically but their occurrence is discarded in next generation.

2.4. Crossover and mutation

The uniform crossover introduced by Syswerda in [28] is chosen. Uniform crossover uses randomly created crossover mask. If crossover mask contains 1 on specific bit position then the offspring's bit on that position will be copied from the first parent. Otherwise, if mask contains 0 on that bit position, then offspring's bit will be copied from the second parent, as shown in Figure 4. A new crossover mask is generated for each pair of parents, with p_{unif} probability bit equals 1, while $1 - p_{\text{unif}}$ probability bit equals 0.

Crossover mask	**1 0 1 1 0 1**
1. parent	**XXXXXX**
2. parent	**YYYYYY**
1. offspring	**XYXXYX**
2. offspring	**YXYYXY**

Figure 4. Uniform crossover.

In our GA implementation, the crossover rate is $p_{\text{cross}} = 0.85$, and probability of uniform crossover is $p_{\text{unif}} = 0.3$. Thus, the crossover is performed for approximately 85% pairs of individuals, and about 30% of bits are exchanged. The differences in experimental results among crossover schemes (uniform, one-point, two-point, multi- point) are smaller than differences among selection schemes, but they are quite visible.

The simple mutation with rate (per bit) $p_{\text{mut}} = 0.005$ is used. To provide faster execution, the simple mutation operator is performed by using a Gausian distribution. Let $N_{\text{mut}} = (N_{\text{pop}} - N_{\text{elite}})^* p_{\text{mut}}$ be an average number of mutations in population and $\sigma^2 = (N_{\text{pop}} - N_{\text{elite}})^* p_{\text{mut}}{}^* (1 - p_{\text{mut}})$ a standard deviation. The number of mutations is generated by a random pick in Gausian $(N_{\text{mut}}, \sigma^2)$ distribution. After that, the positions of mutation sites in the population strings are randomly generated and their number being the same as the number of mutations. Only the muted genes are processed by this method, while the others are not. The number of muted genes is relatively small in comparison to entire population. On this way the run-time performance of a simple mutation operator is improved without changing its nature.

2.5. Parameters

First generation is randomly initialized, because the maximal diversity of the population is maintained in this way. The experiments have been carried out with initialization by some heuristics. In this case, fitness of the first generation got better, but heuristics produced worse gradient in the fitness function of subsequent generations and worse overall results.

Maximal number of generations is $N_{\text{gener}} = 2000$, except in the case of large-scale test instances MT, where is $N_{\text{gener}} = 4000$. The finishing criterion is based on the number of consecutive generations with unchanged best individual. If that number exceeds value N_{repeat}, given in formula (5), execution of GA is stopped.

$$N_{\text{repeat}} = \begin{cases} 2\sqrt{m \cdot n}, & \text{for instances obtained from ORLIB [5]} \\ \sqrt{m \cdot n}, & \text{for generated instance.} \end{cases} \tag{5}$$

It is not possible to prove optimality of the obtained solution by GA. If the optimal solution is known in advance, it can be used for error measurement. In the case when optimal solution is not known in advance, the best-obtained solution by GA is used for error measurement.

2.6. Improving implementation of genetic algorithm by caching

The run-time performance of GA, in our implementation, is also improved by caching. The caching technique decreases run-rime of GA and has no influence on other GA aspects. The caching is used to avoid an attempt to compute the same objective value. During the first computation, objective values are remembered and reused later. If an objective value is computed for a particular string and the same string appears again, the cached values are used to avoid repetition of computing.

Two conditions are necessary for successful applying of caching GA:

- large evaluation time of objective value function;
- significant frequency of same strings in population over the generations.

SPLP has relatively slow function for computing objective value and fulfills the first condition.

Although duplicate strings are discarded from population, the steady-state replacement of generations with elitist strategy provides appearance of some individuals in several subsequent generations. On this way the second condition is also satisfied and caching technique is significant in improving run time performance for our solution of SPLP.

Simple but effective Least Recently Used (LRU) caching strategy is implemented. The LRU strategy is applied through double hash table – a fast and powerful way for caching GA. Caching is performed with cache memory size of 5000 individual strings, *i.e.* approximately 1 MB of main memory. In practice, for all used SPLP test instances, run-time was significantly improved by caching GA. For detailed information about caching GA see [23].

3. Computational results

3.1. Computer environment and test instances

GA is accomplished by a program written in ANSI C. The whole implementation is in character mode display; thus it is portable for all platforms (MS-DOS, UNIX, ...). The program is divided into two parts:

- Kernel of GA, containing common GA functions applicable for various problems;
- Specific GA functions for SPLP, related to: I/O operations, initialization and objective value function.

The problem instances 41–134 and A to C (used in this section) are taken from ORLIB [5]. Parameters of these instances are given in Table 1.

Table 1. Parameters for SPLP instances taken from ORLIB.

Problem instance	Size	File sizes
41 - 44, 51, 61 - 64, 71-74	16×50	10 KB
81 - 84, 91 - 94, 101 - 104	25×50	15 KB
111-114,121-124,131-134	50×50	31 KB
A - C	100×1000	1.2 MB

The randomly generated large-scale SPLP instances MO, MP, MQ, MR, MS and MT are produced by authors. Input parameters for this generator are integers n, m, $b_{\min}$, $b_{\max}$ and real numbers $c_{\min}$, $c_{\max}$, $f_{\min}$, $f_{\max}$. Particular values of those parameters are given in Table 2.

The customer demands b_i are randomly generated from interval $[b_{\min}, b_{\max}]$ and then members of transportation cost matrix, from interval $[c_{\min}, c_{\max}]$ multiplied by b_i, are generated. After that, the sum $S_i = \sum_{j=1}^{n} c_{ij}$ is computed for every facility i. The sum S_i denotes a cumulative cost of all customer demands only from given facility. In the final phase, an inverse scaling into the interval $[f_{\min}, f_{\max}]$ is performed by the following formula:

$$f_i = f_{\max} - \frac{(S_i - S_{\min})\cdot(f_{\max} - f_{\min})}{S_{\max} - S_{\min}} \,. \tag{6}$$

This corresponds to some real situations, where smaller transportation costs c_{ij} produce higher facility establishing cost f_i, and *vice versa*.

The instances, generated by input parameters (shown in Tab. 2), have a small number of useless facility sites (facility sites that have no chance to be established), and a very large number of suboptimal solutions. An implication of that fact is very difficult solving by dual based and other Branch-and-Bound techniques (see Tab. 4). In that case, elimination of useless facility sites is not enough, because there is very large number of suboptimal solutions.

Table 2. Input parameters for generator.

Problem instance	Size	f	c	b	File sizes
MO1-MO5	100×100	50 - 300	2 - 10	1 - 5	100 KB
MP1-MP5	200×200	100 - 600	2 - 10	1 - 5	400 KB
MQ1-MQ5	300×300	150 - 900	2 - 10	1 - 5	900 KB
MR1-MR5	500×500	100 - 600	0.5 - 5	1 - 5	2.4 MB
MS1-MS5	1000×1000	200 - 1200	0.5 - 5	1 - 5	9.5 MB
MT1-MT5	2000×2000	400 - 2400	0.5 - 5	1 - 5	36.3 MB

The testing process is carried out by PC compatible computer AMD 80486DX5 at 133 MHz with 64 MB of memory.

3.2. Experimental results

An effort has been made to compare performances of our GA implementation directly to the other ones, but the only one available is Erlenkotter's Dualoc algorithm described in [13]. We are especially grateful to him for providing his Dualoc code. Two variants of Dualoc are available:

- full implementation for finding optimal solution and
- reduced heuristic that contains only Dual Ascent and Dual Adjustment phase without Branch-and-Bound (BnB).

Both approaches are suitable for solving large-scale problem instances. The results of executing our GA implementation are summarized in Table 3 and can be compared to results of the other two methods given in Tables 4 and 5.

The columns in Table 3 describe:

- names of SPLP instances;
- number of instances in group and number of executions for each of them;
- average number of generations necessary for finishing GA execution;
- average run-time in seconds;
- quality of solutions, *i.e.* number of tests with optimal solution (or the best-obtained solution), and with relative error less than: 0.2%, 1% and over 1%.

The optimal solution is not known for some SPLP instances (MR1, MR2, MS1-MS5 and MT1-MT5). In that case the best-obtained solution is reported instead of optimal one. In the Tables 2–4 optimal solutions are noted by Opt, while the best-obtained solutions are noted with BS.

Since genetic operators: selection, crossover and mutation are undeterministic, every problem instance should be run multiple times and an average value is computed.

Tables 4 and 5 show names of problem instances in similar way, as well as number of runs, average number of Dualoc iterations, run-time in seconds, and average error of result compared to optimal solution. Note that original version of Dualoc always produces optimal solution if it finishes its execution regularly. In some SPLP instances (MS) execution is truncated after some time (about 200 min)

TABLE 3. Results of GA.

Instance	Num. of runs	Avg. gener.	Avg. time (s)	Opt. (BS)	Err. <0.2%	Err. <1%	Err. >1%
41 – 74	13×20	74.1	0.86	260	-	-	-
81 – 104	12×20	104.6	1.26	240	-	-	-
111-134	12×20	199.2	2.97	198	40	2	-
A – C	3×20	1 015	83.1	42	9	9	-
MO	5×20	216.4	5.49	93	7	-	-
MP	5×20	387.5	16.60	100	-	-	-
MQ	5×20	571.4	34.97	100	-	-	-
MR	5×20	917.6	93.69	99	0	1	-
MS	5×20	1 844	379.6	100	-	-	-
MT	5×20	3 612	1812.3	100	-	-	-

TABLE 4. Results of DUALOC.

Instance	Num. of runs	Avg. iterations	Avg. time (s)	Avg. error
41 – 74	13×1	1	<0.01	Opt
81 – 104	12×1	4	<0.01	Opt
111 – 134	12×1	3	<0.01	Opt
A – C	3×1	1 223	25.173	Opt
MO	5×1	26 268	32.54	Opt
MP	5×1	126 395	369.71	Opt
MQ	5×1	499 011	2913.7	Opt
MR1	1	10 307 844	99 424	9.4% to BS
MR2	1	7 050 800	68 008	6.7% to BS
MR3	1	5 522 898	54 167	Opt
MR4	1	8 121 349	79 779	Opt
MR5	1	8 375 395	78 444	Opt
MS	5×1	288 315	12 279	11.32% to BS

because it is too long and the obtained results are quite bad. In that case the DUALOC produces solutions with more than 10% error in related to BS solutions obtained by GA.

All presented results of DUALOC in Table 4 are obtained through maximum/one-pass dual improvement. The application is accomplished through the maximum improvement dual adjustment procedure at the initial (BnB) node and only once applied at subsequent nodes. The testing of DUALOC, with maximum dual improvement at all BnB nodes is also performed, but the obtained results are considerably worse and they are not presented. For detailed description of these variants of DUALOC see [13] and [21].

TABLE 5. Results of DUALOC without BnB.

Instance	Num. of runs	Avg. iterations	Avg. time (s)	Avg. error
41 - 74	13×1	1	<0.01	Opt
81 - 104	12×1	4	<0.01	Opt
111 - 134	12×1	2.8	0.013	Opt
A - C	3×1	335.6	6.5	5.74% to Opt
MO	5×1	214.4	0.352	9.74% to Opt
MP	5×1	695.4	2.74	9.63% to Opt
MQ	5×1	1 450	9.8	10.4% to Opt
MR	5×1	2 886	42.61	13.35% to Opt (BS)
MS	5×1	9 625	348.8	15.29% to BS

The results of testing for reduced DUALOC version are given in Table 5. This heuristic contains only Dual Ascent and Dual Adjustment phase, without branching (BnB) phase. This method has a short run-time, but produces only a suboptimal solution, just like GA.

For MR problem instances optimal solution is known in some cases (MR3-MR5) and solution quality, for the reduced version of DUALOC, is calculated according to it. In other cases (MR1-MR2), the BS solutions obtained by GA are used for the calculation of solution quality.

Also, the optimal solutions (best-obtained solutions) have not been produced by GA in every running. Suboptimal solutions are obtained with relative error smaller than 1%. According to Table 3, the number of appearance of suboptimal solutions is less than 1/3.

3.3. COMPARISON OF RESULTS

As we can see from Tables 3–5, both versions of DUALOC are perfect for instances 41–134. The optimal solutions are obtained only in a few iterations. The execution time of GA implementation is approximately 80–300 times worse than DUALOC. This is a result obtained for the problem instances of relatively small size or with large number of useless facility sites.

For instances A–C, the original DUALOC is still better than GA, but the difference in run-time is significantly smaller (25.17 *vs.* 83.1 s). The reduced DUALOC quickly produces results, but quality of the obtained suboptimal solutions is not so good. The relative error is large in comparison to optimal solution (5.74%).

For all subsequent instances (MO-MS), the reduced DUALOC gives worse quality of solutions, with errors of 9.5% to 15.5%. That level of quality is unacceptable for practical purposes. Because of that the reduced DUALOC is omitted in the next comparisons. By increasing the instance size, run-time of reduced DUALOC increases considerably faster than GA run-time.

In the case of instances MO-MQ with medium size and a small number of useless facility sites, the original Dualoc and GA produce optimal solutions, but run-time of GA is 5–80 times better. Speedup is more than 500 times for MR instances and we believe that speedup for instances MS and MT increases exponentially with instance size.

The obtained result might be explained by presenting nature of the GA and Dualoc.

The Dualoc algorithm performs BnB search and produces the optimal solution having an exponential complexity. The Dualoc algorithm is useful for the problem instances with relatively small number of suboptimal solutions near the optimal one. In that case, the useless facility sites are eliminated implicitly and the search space is reduced only on small number of useful facility sites. For that reason the solution is obtained very quickly. However, for the problem instances with a small number of useless facilities and a large number of suboptimal solutions, the search space is very large. Therefore, Dualoc makes small improvement in every BnB iteration with large overall run-time.

GA is a robust technique for solving NP-hard problems (see [10]) and large number of suboptimal solutions is a good stepping-stone to reach optimal value. The individuals of a population quickly obtain near optimal values and by their recombination the optimal solution is reached. On the contrary, a small number of near optimal solutions implies a slow producing of individuals near to optimal values. After a lot of iterations, only few individuals have near-optimal values and it is difficult to obtain optimal solution by recombination.

Similarly to Körkel's PDLOC memory requirement, GA implementation is about 2 times smaller than Dualoc. For example, for 1000×1000 instances GA implementation allocated approximately 12 MB memory, but Dualoc allocated 24MB.

3.4. Comparison with the other implementations

The other implementations, except Dualoc, have not been available for direct performance comparison. Indirectly, we can make some conclusions according to the accessible information in particular papers.

Quite different techniques for solving SPLP are demonstrated by algorithms used in [7, 18, 19]. The experimental results presented in these papers are better than Dualoc (in some cases), but the speedup is small. In the papers [7,18], only results for SPLP instances with small size (up to 100×100) are presented.

From [2] one can see that simulated annealing method produces good solutions, but in run-time significantly longer than the other methods.

Körkel's PDLOC algorithm, described in [21], is more recent and more sophisticated than Dualoc. This commercial product practically has been unavailable for us. Adopting the test results shown in [21], we can conclude that PDLOC generates solutions 10–100 times faster than Dualoc.

4. Conclusion

A GA implementation for solving simple plant location problem is explored. During the testing process, experiments are carried out with several variants of selection and crossover. We conclude that rank-based selection and uniform crossover provided the best performances. Binary representation of facility sites provides the successful performances of GA. The objective value function is efficiently implemented and strengthened by caching GA.

Our implementation is especially convenient for SPLP instances with more than 1000 facility locations and customers. This approach is recommended for the large-scale problem instances with the small number of useless facility sites. For such instances the other methods for solving SPLP have poor performances, *i.e.* only the GA implementation produces qualitative solutions in reasonably short runtime.

The research presented in this paper can be improved, generalized and extended in several directions:

- testing of our approach for larger size problem instances (more than 2000 facilities and customers) on powerful computers;
- hybridizing of GA with the other methods for solving SPLP;
- upgrading of used genetic algorithm with ability to verify optimal solution;
- parallelizing of the genetic algorithm for distributed and multiprocessor systems;
- applying our approach to the other related location problems: p-median, capacitated facility location, dynamic facility location, multi-product uncapacitated facility location and data file location/allocation problems.

Special thanks are due to Donald Erlenkotter, for providing his Dualoc code for performance comparison.

References

[1] C.H. Aikens, Facility Location Models for Distribution Planning. *European J. Oper. Res.* **22** (1985) 263-279.

[2] M.L. Alves and M.T. Almeida, Simulated Annealing Algorithm for the Simple Plant Location Problem: A Computational Study. *Rev. Invest.* **12** (1992).

[3] A.A. Aqeev, V.S. Beresnev, Polynomially Solvable Cases of the Simple Plant Location Problem, in *Proc. of the First Integer Programming and Combinatorial Optimization Conference*, edited by R. Kannan and W.R. Pulleyblank. University of Waterloo Press, ON, Canada (1990) 1-6.

[4] D. Beasley, D.R. Bull and R.R. Martin, An Overview of Genetic Algorithms. *Univ. Computing* **15** (1993) 170-181.

[5] J.E. Beasley, Obtaining Test Problems *via* Internet. *J. Global Optim.* **8** (1996) 429-433, `http://mscmga.ms.ic.ac.uk/info.html`

[6] J.E. Beasley, Lagrangean Heuristic for Location Problems. *European J. Oper. Res.* **65** (1993) 383-399.

[7] A.R. Conn and G. Cornuejols, A Projection Method for the Uncapacitated Facility Location Problem. *Math. Programming* **46** (1990) 273-298.

[8] G. Cornuejols, G.L. Nemhauser and L.A. Wolsey, The Uncapacitated Facility Location Problem, in *Discrete Location Theory*, edited by P.B. Mirchandani and R.L. Francis. John Wiley & Sons (1990), Chapter 3, pp. 120-171.

[9] P.M. Dearing, Location Problems. *Oper. Res. Lett.* **4** (1985) 95-98.

[10] K.E. De Jong and W.M. Spears, Using Genetic Algorithms to Solve NP-Complete Problems, in *Proc. of the Third International Conference on Genetic Algorithms.* Morgan Kaufmann, San Mateo, CA (1989) 124-132.

[11] C. De Simone and C. Mannino, *Easy Instances of the Plant Location Problem*, Technical Report R-427. Gennaio, University of Roma, Italy (1996).

[12] R. Dorne and J.K. Hao, A New Genetic Local Search Algorithm for Graph Coloring, in *Proc. of the 5th Conference on Parallel Problem Solving from Nature – PPSN V.* Springer-Verlag, *Lecture Notes in Comput. Sci.* **1498** (1998) 745-754.

[13] D. Erlenkotter, A Dual-Based Procedure for Uncapacitated Facility Location. *Oper. Res.* **26** (1978) 992-1009.

[14] R.L. Francis, L.F. McGinnis and J.A. White, Locational Analysis. *European J. Oper. Res.* **12** (1983) 220-252.

[15] B. Freisleben and P. Merz, New Genetic Local Search Operators for the Traveling Salesman Problem, in *Proc. of the 4th Conference on Parallel Problem Solving from Nature – PPSN IV.* Springer-Verlag *Lecture Notes in Comput. Sci.* **1141** (1996) 890-899.

[16] L.L. Gao, E. Robinson and Jr. Powell, Uncapacitated Facility Location: General Solution Procedure and Computational Experience. *European J. Oper. Res.* **76** (1994) 410-427.

[17] V.P. Grishukhin, On Polynomial Solvability Conditions for the Simplest Plant Location Problem, in *Selected topics in discrete mathematics*, edited by A.K. Kelmans and S. Ivanov. American Mathematical Society, Providence, RI (1994) 37-46.

[18] M. Guignard, A Lagrangean Dual Ascent Algorithm for Simple Plant Location Problems. *European J. Oper. Res.* **35** (1988) 193-200.

[19] K. Holmberg, *Experiments with Primal-Dual Decomposition and Subgradient Methods for the Uncapacitated Facility Location Problem*, Research Report LiTH-MAT/OPT-WP-1995-08, Optimization. Department of Mathematics, Linkoping Institute of Technology, Sweden (1995).

[20] S. Khuri, T. Back and J. Heitkotter, An Evolutionary Approach to Combinatorial Optimization Problems, in *Proc. of CSC'94.* Phoenix, Arizona (1994).

[21] M. Koerkel, On the Exact Solution of Large-Scale Simple Plant Location Problems. *European J. Oper. Res.* **39** (1989) 157-173.

[22] J. Krarup and P.M. Pruzan, The Simple Plant Location Problem: Survey and Synthesis. *European J. Oper. Res.* **12** (1983) 36-81.

[23] J. Kratica, Improving Performances of the Genetic Algorithm by Caching. *Comput. Artificial Intelligence* **18** (1999) 271-283.

[24] P. Merz and B. Freisleben, A Genetic Local Search Approach to the Quadratic Assignment Problem, in *Proc. of the Seventh International Conference on Genetic Algorithms.* Morgan Kaufmann (1997) 465-472.

[25] P. Merz and B. Freisleben, Genetic Local Search for the TSP: New Results, in *Proc. of the 1997 IEEE International Conference on Evolutionary Computation.* IEEE Press (1997) 159-164.

[26] C. Ryu and M. Guignard, *An Exact Algorithm for the Simple Plant Location Problem with an Aggregate Capacity Constraint*, TIMS/ORSA Meeting. Orlando, FL (1992) 92-04-09.

[27] H.P. Simao and J.M. Thizy, A Dual Simplex Algorithm for the Canonical Representation of the Uncapacitated Facility Location Problem. *Oper. Res. Lett.* **8** (1989) 279-286.

[28] G. Syswerda, Uniform Crossover in Genetic Algorithms, in *Proc. of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, San Mateo, CA (1989) 2-9.

[29] D. Whitley, The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best, in *Proc. of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, San Mateo, CA (1989) 116-123.