View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Computing and Informatics (E-Journal - Institute of Informatics, SAS, Bratislava)

Computing and Informatics, Vol. 29, 2010, 183–201

AN EFFICIENT GENETIC ALGORITHM FOR SOLVING THE MULTI-LEVEL UNCAPACITATED FACILITY LOCATION PROBLEM

Miroslav Marić

Faculty of Mathematics, University of Belgrade
Studentski trg 16/IV
11 000 Belgrade, Serbia
e-mail: maric.m@sbb.rs

Manuscript received 1 February 2008; revised 20 May 2008
Communicated by Vladimír Kvasnička

Abstract. In this paper a new evolutionary approach for solving the multi-level uncapacitated facility location problem (MLUFLP) is presented. Binary encoding scheme is used with appropriate objective function containing dynamic programming approach for finding sequence of located facilities on each level to satisfy clients' demands. The experiments were carried out on the modified standard single level facility location problem instances. Genetic algorithm (GA) reaches all known optimal solutions for smaller dimension instances, obtained by total enumeration and CPLEX solver. Moreover, all optimal/best known solutions were reached by genetic algorithm for a single-level variant of the problem.

Keywords: Facility location, genetic algorithms, evolutionary approach

1 INTRODUCTION

The past five decades have witnessed an expansive growth in the field of facility location area. Much research has been carried out on location problems which require minimization of physical distance, total travel time, or some other related cost, and it is often assumed that facilities are sufficiently large to meet any demand likely to be encountered. There are several deterministic uncapacitated models proposed in the literature so far.

The uncapacitated facility location problem (UFLP), also known in the literature as the simple plant location problem and the uncapacitated warehouse location problem, is one of the fundamental and most studied models in facility location theory. The objective is to minimize the sum of fixed costs and transportation costs in order to satisfy given demands from a set of clients. This is realized by selecting potential facility locations on a network from a given set. Presentation of all relevant methods is out of this paper's scope. Several survey papers are [11, 21, 37, 44].

Some recent successful methods for solving UFLP proposed in the literature up to now are: genetic algorithms [15, 25, 26], variable neighborhood search [20], tabu search [42], simulated annealing [45], Lagrangean relaxation [9], filter and fan [18, 19], hybrid multistart heuristics [36] and particle swarm optimization [38]. There are also methods for solving some generalizations of the basic problem: multi-objective UFLP [32, 43] and dynamic UFLP [12, 13].

Contrary to previous single-level case, the multi-level version of problem is considered only in few papers: ([1, 2, 3, 6, 14, 46]). However, except of [14], all other papers are theoretical without experimental results. In [14] four methods for solving the MLUFLP are implemented:

- the linear program solution rounding 3-approximation algorithm (MLRR), by Aardal, Chudak and Shmoys [1],
- the path reduction of the k-level facility location problem to a single-level problem (PR-RR), by Chudak and Shmoys [8],
- the local improvement 3-approximation algorithm for the path reduction (PR-LI), by Charikar and Guha [7],
- the facility cost oblivious shortest path algorithm (SP), by Edwards [14].

The first three algorithms are based on linear programming relaxation of the model, described in Section 2, which has enormous number of variables and moderate number of constrains. Therefore, all three algorithms are capable of solving only small size MLUFLP instances with up to 52 potential facility locations. Gaps from LP lower bounds are up to 98.5 % for MLRR, up to 3.3 % for PR-RR and up to 10.8 % for PR-LI. Running times of these methods are: up to 813 seconds for MLRR, up to 183 seconds for PR-RR, up to 105 seconds for PR-LI. Gap values are relatively satisfiable, except for MLRR, but the corresponding running times are very large even for these small size instances. Obviously, all three linear programming based methods are unable to solve larger size problem instances. On the other hand, SP produce results very quickly (up to 0.07 seconds), but the gaps are very large – up to 732 %. Therefore, none of these algorithms is capable of solving real medium size and large-scale MLUFLP instances.

The paper is organized as follows. In Section 2 mathematical formulation is presented. Dynamic programming approach is described in Section 3, which is the main contribution of this paper that enabled GA to solve large-scale problem instances. The next two sections contain main features of the GA implementation designed for the MLUFLP and computational results on various instances.

2 MATHEMATICAL FORMULATION

The input data to the MLUFLP consists of a set of facilities F ($|F| = m$) partitioned into k levels, denoted $F_1, \ldots, F_k$, a set of clients D ($|D| = n$), a fixed cost f_i for establishing facility $i \in F$, and a metric that defines transportation costs c_{ij} for each $i, j \in F \cup D$. A feasible solution assigns each client a sequence of k facilities, one from each level $F_k, \ldots, F_1$, respectively. A feasible solution is charged the sum of the fixed costs of the facilities used, plus the transportation costs of the clients' assignments. Each client's transportation cost is the sum of transportation cost from itself to the first facility of its sequence, plus the transportation cost between successive facilities of its sequence. An optimal solution to the multi-level uncapacitated facility location problem is a feasible solution of minimum total cost.

The multi-level uncapacitated facility location problem is NP-hard, since it is described as a generalization of the uncapacitated facility location problem that is proved to be NP-hard in [23].

The integer programming formulation of the multi-level uncapacitated facility location problem from [14] is used in this paper. The assignment of a client $j \in D$ to a valid sequence of facilities can now be represented as the assignment of j to a path p from j to one of the top level facilities F_1. The set of all valid sequences of facilities is defined by $P = F_k \times \ldots \times F_1$ and the transportation cost of client j's assignment to sequence $p = (i_k, \ldots, i_1)$ by $c_{pj} = c_{ji_k} + c_{i_k i_{k-1}} + \ldots + c_{i_2 i_1}$. Variable y_i has value of 1 if facility i is established and 0 otherwise. Similarly, variable x_{pj} represents whether or not a client j is assigned to the path p. Using the notation mentioned above, the problem can be written as:

$$\min \sum_{i \in F} f_i y_i + \sum_{p \in P} \sum_{j \in D} c_{pj} x_{pj} \tag{1}$$

$$\sum_{p \in P} x_{pj} = 1, \qquad \text{for each } j \in D, \tag{2}$$

$$\sum_{p \ni i} x_{pj} \leq y_i, \qquad \text{for each } i \in F, j \in D, \tag{3}$$

$$x_{pj} \in \{0, 1\}, \qquad \text{for each } p \in P, j \in D, \tag{4}$$

$$y_i \in \{0, 1\}, \qquad \text{for each } i \in F. \tag{5}$$

The objective function (1) minimizes the sum of overall transportation cost and fixed costs for establishing facilities. Constraint (2) ensures that every client is assigned to a path while constraint (3) guarantees that any facility on a path used by some client is paid for. Constraints (4) and (5) reflect binary nature of variables x_{pj} and y_i.

Example 1. An example of the MLUFLP is shown below. It assumes $k = 2$ levels of $m = 6$ facilities: the first level contains 2 potential facilities and the second

4 potential facilities. There are $n = 5$ clients to be served in this example. The fixed costs of establishing facilities are given in Table 1, the distances between facilities of different levels and the distances between clients and facilities on second level are given in Table 2 and Table 3, respectively.

Facilities	f1	f2	f3	f4	f5	f6
Fixed cost	20	20	10	10	10	10

Table 1. Fixed costs

	f1	f2
f3	12	13
f4	11	15
f5	16	12
f6	12	15

Table 2. Distance between facilities on level 1 and level 2

	f3	f4	f5	f6
client 1	5	2	4	3
client 2	4	1	6	8
client 3	1	5	2	3
client 4	8	1	5	1
client 5	4	9	2	1

Table 3. Distance between clients and facilities

The total enumeration technique, described in Section 5, is used to obtain optimal solution. Established facilities are: f1 on the first and f4, f6 on the second level. The objective function value is 105. The sequences of facilities for each client are shown in Table 4.

3 DYNAMIC PROGRAMMING APPROACH

Dynamic programming is very popular method for solving combinatorial optimization problems with optimal substructure of solutions. The optimal substructure means that optimal solutions of subproblems can be used to find the optimal solutions of the overall problem. Polynomial number of states and steps are very important for successful application. Obviously, this is fulfilled only if the problem has polynomial complexity.

The main idea in solving the MLUFLP is to decrease the number of potential paths for objective function calculation. Existing methods for the MLUFLP considered all possible paths, which was both time and memory consuming, so they were unable to solve practical size problems.

	level 2	level 1
client 1	f4	f1
client 2	f4	f1
client 3	f6	f1
client 4	f4	f1
client 5	f6	f1

Table 4. Sequences of facilities for clients

Consider a subproblem of the MLUFLP, named the FixedMLUFLP, obtained from the MLUFLP by fixing established facilities, with at least one established facility existing on every level. The MLUFLP is NP-hard, but the FixedMLUFLP has polynomial complexity. Indeed, the FixedMLUFLP has optimal substructure of solutions which will be proved in the following proposition.

Proposition 1. The FixedMLUFLP can be polynomially reduced to shortest path problem in directed acyclic graph (DAG).

Proof. Construct graph G $= < V, E >$ in the following way:

- Let $V = F \bigcup D$
- Let $(u, v) \in E, u, v \in F$ if and only if $u \in F_l$, $v \in F_{l+1}$, $y_u = y_v = 1$, for $l \in \{1, 2, \ldots, k-1\}$ and,
- $(u, v) \in E, u \in F, v \in D$ if and only if $u \in F_k, y_u = 1$
- Weight on edge (u,v) is defined as a distance between u and v.

Constructed graph G is obviously acyclic, since its edges connect vertices on levels l and $l+1$, or the last level k and clients. On graph G, for every client exist paths from the first level facilities to it, because in the FixedMLUFLP on each level at least one established facility exists and all facilities from subsequent levels are connected. Evidently, for every client j its shortest path p from facilities on level 1 to j holds $x_{pj} = 1$. Consequently, objective value of the FixedMLUFLP can be calculated by summing the length of these shortest paths for all clients j and adding fixed costs. Therefore, the FixedMLUFLP can be polynomially reduced to shortest path problem in directed acyclic graph. □

Since the shortest path problem in directed acyclic graph can be optimally solved by dynamic programming, it is obvious that the dynamic programming is also applicable for the FixedMLUFLP.

Dynamic programming approach can be used in following way. The array of mininimal costs (named cs) is established and contains transportation cost of established facilities from facilities on the first level. Clients are included as facilities on the $(k+1)^{\text{st}}$ level, so the array cs consists of $m+n$ members.

In the beginning, the array cs is initialized to 0 on positions which correspond to facilities on the first level and ∞ on other positions. Next, for all levels $l =$

$2, 3, \ldots, k+1$ and all established facilities on that level, i.e. $i \in F_l$ and $y_i = 1$, the array cs is defined recursively by (6).

$$cs[i] = \min_{k \in F_{l-1} \wedge y_k = 1} (cs[k] + c_{ik}). \tag{6}$$

Finally, the objective value of the FixedMLUFLP is

$$obj_{ind} = \sum_{i \in F} f_i y_i + \sum_{i=m+1}^{m+n} cs[i]. \tag{7}$$

It is obvious that the number of states (members of array cs) is $m + n$. The complexity of (6) is $O(n \cdot \max_i(num[i])), 1 \leq i \leq k$, where array num carries information about the number of facilities on each level. In the worst case of $k = 1$, the complexity is $O(n^2)$.

4 PROPOSED GA METHOD

Genetic algorithms (GAs) are stochastic search techniques that imitate some spontaneous optimization processes in the natural evolution. At each iteration, GA works with a set of individuals, named population. Each individual in the population represents an encoded solution of a problem. The initial population is either randomly or heuristically generated. The individuals' quality in the current population is evaluated by using a fitness function. Good individuals are selected to produce new generation, by applying genetic operators crossover and mutation. The new individuals – offsprings – replace some of the individuals from the current population. The described process is iteratively preformed until some stopping criterion is satisfied. Detailed description of GAs can be found in [33]. Extensive computational experience on various optimization problems shows that GA often produces high quality solutions in a reasonable time [17, 22, 27, 30, 31, 34, 35, 39, 40]. Moreover, GA has shown to be robust with respect to parameter choice in reasonable bounds on quite different problems [10, 15, 16, 22, 27, 28, 29, 39, 40, 41].

The outline of our GA implementation is given below, where N_{pop} denotes the overall number of individuals in the population, N_{elite} is a number of elite individuals and ind and obj_{ind} are the individual and its objective value.

```
Input_Data();
Population_Init();
while not Stopping_Criterion() do
    for ind:= (N_elite + 1) to N_pop do
      if (Exist_in_Cache(ind)) then
        obj_ind:= Get_Value_From_Cache(ind);
      else
        obj_ind:= Objective_Function(ind);
         Put_Into_the_Cache_Memory(ind,obj_ind);
```

```
        if (Full_Cache_Memory()) then
          Remove_LRU_Block_From_Cache_Memory();
        endif
      endif
    endfor
    Fitness_Function();
    Selection();
    Crossover();
    Mutation();
endwhile
Output_Data();
```

The binary encoding of the individuals used in this implementation is as follows. The set of potential facilities F is naturally represented in the individual by a binary string of length m. Digit 1 at the i^{th} place of the string denotes that $y_i = 1$, while 0 shows the opposite ($y_i = 0$).

Fixing the established facilities obtained from genetic code, objective value can be calculated by solving the FixedMLUFLP. Since in the FixedMLUFLP has to be at least one established facility, the array of established facilities is checked for that property. If the array holds that property, objective function is evaluated by dynamic programming approach described in previous section. Otherwise, the solution is not valid and the individual is marked as unfeasible.

Example 2. Let genetic code is 100101. Then $y_1 = y_4 = y_6 = 1$ and $y_2 = y_3 = y_5 = 0$. That genetic code represents the optimal solution from Example 1.

The Objective_Function(ind), for feasible individual ind, is evaluated in four steps.

1. In the first step, the values of variables y_i are obtained from the genetic code. The running time complexity of this step is $O(m)$.
2. In the second step, the array of mininimal costs cs is initalized. As can be seen from previous section, array cs carries information about total minimal costs for serving clients and facilities (except the ones on the first level), regarding the costs for serving facilities on the upper level. The minimal cost values in cs, for the facilities on the first level are initially set to zero, while the costs for facilities on remaining levels and clients are set to a large constant $INF = 10^{30}$. The time complexity for initalizing array cs is $O(n + m)$.
3. The minimal costs for each client and each facility are calculated by dynamic programming approach defined in the previous section. For each facility in each level (except the first one), the array cs (initialized in the second step) is updated. The minimal cost value for each facility is the minimum of the sum of the minimal cost for established facility on the upper level and transportation cost between the two facilities, as defined in (6). The same procedure is done for each client: among the established facilities from the last level, the one with the

minimal sum of the corresponding minimal cost value and the transportation cost facility-client is taken. The time complexity of this step, in worst case of $k = 1$, is $O(n^2)$.

4. Finally, the running time complexity for calculating the objective value $O(n+m)$.

From the explanations given above, it is obvious that the overall time complexity is $O(m + n^2)$.

In Fitness_Function(), the fitness f_{ind} of individual ind $1, 2, ldots, N_{pop}$ is computed by scaling objective values obj_{ind} of all individuals into the interval $[0, 1]$, so that the best individual ind_{min} has fitness 1 and the worst one ind_{max} has fitness 0. More precisely, $f_{ind}\frac{obj_{ind_{max}}-obj_{ind}}{obj_{ind_{max}}-obj_{ind_{min}}}$. The next step is to arrange individuals in non-increasing order of their fitness: $f_1 \geq f_2 \geq \ldots \geq f_{N_{pop}}$.

Usually GAs have relatively small number of elite individuals, because they have a chance to pass into the next generation twice: once through selection operator and once as elite individuals. Such common practice is not adequate for this purpose. In order to obtain satisfactory results of the GA implementation, it is necessary to provide sufficient number of elite individuals to preserve good solutions for exploitation as well as sufficient number of non-elite individuals for exploration. To prevent an undeserved domination of N_{elite} elite individuals over the population, their fitness are decreased by the next formula:

$$f_{ind} = \begin{cases} f_{ind} - \overline{f}, & f_{ind} > \overline{f} \\ 0, & f_{ind} \leq \overline{f} \end{cases}; \qquad 1 \leq ind \leq N_{elite}; \quad \overline{f} = \frac{1}{N_{pop}} \sum_{ind=1}^{N_{pop}} f_{ind}. \tag{8}$$

In this way, even non-elite individuals preserve their chance to pass in the next generation. The described approach allows high elitism without too high selection pressure, which may lead to over-exploitation in the algorithm.

The elitist strategy is applied to N_{elite} elite individuals, which are directly passing to the next generation. The genetic operators are applied to the rest of the population (N_{nnel} $N_{pop} - N_{elite}$ non-elite individuals). The objective value of elite individuals are the same as in the previous generation, so they are calculated only once, providing significant time-savings.

Duplicated individuals, i.e. individuals with the same genetic code are redundant. In order to prevent them to enter the next generation their fitness values are set to zero, except for the first occurrence. Individuals with the same objective value, but different genetic codes, may in some cases dominate in the population by number, which implies that the other individuals with potentially good genes are rare. For that reason, it is useful to limit the number of their appearance to a constant N_{rv}. This is a very effective technique for saving the diversity of the genetic material and keeping the algorithm away from a premature convergence. It consists of two steps for every individual in the population:

Step 1: Check whether the genetic code of the current individual *ind* is identical with the genetic code of any of the individuals from 1 to $ind - 1$. If the answer is positive, set the fitness of *ind* to 0. Otherwise go to **Step 2**;

Step 2: Count the number of the individuals from 1 to $ind - 1$ which did not get fitness 0 in **Step 1** and which have the same objective value as *ind*. If it is greater than or equal to N_{rv}, set the fitness of *ind* to 0.

The selection operator chooses the individuals that will produce offspring in the next generation, according to their fitness. Low fitness-valued individuals have less chance to be selected than high fitness-valued ones. In the standard tournament scheme, one tournament is performed for every non-elitist individual. The tournament size is a given parameter and tournament candidates are randomly chosen from the current population. Only the winner of the tournament, i.e. a tournament candidate with the best fitness, participates in the crossover. The tournament is performed N_{nnel} times on the set of all N_{pop} individuals in the population to choose the N_{nnel} parents for crossover. The same individual from the current generation may participate in several tournaments. The standard tournament selection uses an integer tournament size, which in some cases may be a limiting factor.

The improved tournament selection operator, also known as the fine-grained tournament selection – FGTS [15], is implemented in Selection(). This operator uses a real (rational) parameter F_{tour} which denotes the desired average tournament size. The first type of tournaments is held k_1 times and its size is $\lfloor F_{tour} \rfloor$, while the second type is performed k_2 times with $\lceil F_{tour} \rceil$ individuals participating, so $F_{tour} \approx \frac{k_1 \cdot \lfloor F_{tour} \rfloor + k_2 \cdot \lceil F_{tour} \rceil}{N_{nnel}}$.

Extensive numerical experiments in [15, 16, 17, 39] performed for different optimization problems indicate that FGTS gives the best results for $F_{tour} = 5.4$. For that reason, the same value is used as reasonable choice in this GA implementation. The running time for FGTS operator is $O(N_{nnel} \cdot F_{tour})$. In practice F_{tour} and N_{nnel} are considered to be constant (not depending on n); that gives a constant running time complexity. For detailed information about FGTS see [16].

In Crossover() all non-elitist individuals chosen to produce offsprings for the next generation are randomly paired for exchanging genes in $\lfloor N_{nnel}/2 \rfloor$ pairs. After a pair of parents is selected, a crossover operator is applied to them producing two offsprings. The standard one-point crossover operator is used in this GA implementation. This operator is performed by exchanging segments of two parents' genetic codes starting from a randomly chosen crossover point. The crossover operator is realized with probability $p_{cross} = 0.85$. It means that approximately 85 % pairs of individuals exchange their genetic material.

The standard simple mutation operator is implemented in the proposed GA. It is performed by changing a randomly selected gene in the genetic code of the individual, with a certain mutation rate. During the GA execution it may happen that all individuals in the population have the same gene on a certain position. This gene is called frozen. If the number of frozen genes is l, the search space becomes 2^l times smaller and the possibility of a premature convergence increases rapidly.

The crossover operator can not change the bit value of any frozen gene and the basic mutation rate is often too small to restore lost subregions of the search space. On the other hand, if the basic mutation rate is increased significantly, a genetic algorithm becomes a random search.

For that reason, the simple mutation operator used in Mutation() is modified so that mutation rate is increased only on frozen genes. In this implementation, the mutation rate for frozen genes is increased $2.5 \cdot (1.0/n)$, compared to non-frozen ones $(0.4/n)$. In each generation, we determine positions where all individuals have a given gene fixed and define them as frozen genes. Obviously the set of frozen genes is not fixed, i.e. it may change during the generations.

The initial population is randomly generated in Population_Init(). This approach provides the maximal diversity of the genetic material. This function also computes values of all the individuals of the population.

In order to obtain satisfactory results of our GA implementation, it is necessary to have sufficient number of elite individuals to preserve good solutions for the exploitation, as well as sufficient number of non-elite individuals for the exploration. The population size of $N_{pop} = 150$ individuals with $N_{elite} = 100$ elite and $N_{nnel} = 50$ non-elite individuals is a good compromise between exploitation and exploration part of GA search. In this case, the corresponding values of k_1 and k_2 in FGTS operator are 20 and 30, respectively. The maximal allowed number of individuals with the same objective value is $N_{rv} = 40$.

The run-time performance of GA is improved by using caching technique. The main idea is to avoid computing the same objective value every time when genetic operators produce individuals with the same genetic code. Evaluated objective values are stored in a hash-queue data structure using the least recently used (LRU) caching technique. When the same code is obtained again, its objective value is taken from the cache memory that provides time-savings. In this implementation the number of individuals stored in the cache memory is limited to 5 000. Function Exist_in_Cache(*ind*) investigates whether the cache memory contains the individual *ind*. In that case objective value obj_{ind} is directly taken from the cache memory by Get_Value_From_Cache(*ind*). Otherwise, the objective value obj_{ind} is calculated and the pair (*ind*, obj_{ind}) is stored in the cache memory by Put_Into_the_Cache_Memory(*ind*, obj_{ind}). If the cache memory is full, in order to make space for the new entry, the function Remove_LRU_Block_From_Cache_Memory() removes the block (*ind*, obj_{ind}), which has been the least recently used. For detailed information about caching GA see [24].

5 EXPERIMENTAL RESULTS

In order to verify the optimality of GA solutions for small size instances, a total enumeration technique has been developed. This technique simply checks all subsets of F and computes minimal objective value of the MLUFLP by using the same objective function as GA (see Section 4). Since all possible subsets of F are

examined, the obtained minimal objective value is obviously the optimal solution of current MLUFLP instance. Additionally, an integer programming formulation is implemented and tested by in CPLEX 8.1.0 solver in order to obtain optimal solutions on *cap*131 MLUFLP instances.

In this section the computational results of the GA method are presented. All tests were carried out on an Intel 1.8 GHz with 512 MB memory. The algorithms were coded in C programming language.

The GA is tested on the instances from the Imperial College OR library (ORLIB) [4, 5]. This data set contains instances for a variety of operations research problems, including the uncapacitated facility location problem. However, these instances are designed only for single-level UFLP and contain small or medium number of potential facilities. The large-scale M* instances from [35] are used to test the GA performance on practical size problems. The instances from [14] are not available and they are, in most cases, based on standard ORLIB instances.

This paper is considering instances that are generated from ORLIB in similar way as in [14]. Since ORLIB instances contain only client-facility distances, it was necessary to generate facility-facility distances out of client-facility distances while preserving triangle inequalities. Each facility-facility distance has to be less than or equal to the sum of facility-client and client-facility distances for each client(named Su_{client}). Similarly, each facility-facility distance has to be greater than or equal to absolute value of difference of facility-client and client-facility (named Di_{client}). The author in [14] considered only the first condition by using minimum of the facility-client and client-facility sum, i.e. $min_{client}(Su_{client})$. This paper has adopted the formula $\frac{min_{client}(Su_{client})+max_{client}(Di_{client})}{2}$, that reflects more realistic assumption in practice. The same technique is applied on M* instances for UFLP from [26] to generate challenging large-scale MLUFLP instances for the presented GA.

The finishing criterion of GA is the maximal number of generations $N_{gen} = 5\,000$. The algorithm also stops if the best individual or best objective value remains unchanged through $N_{rep} = 2\,000$ successive generations. Since the results of GA are nondeterministic, the GA was run 20 times on each problem instance. In order to avoid the presentation of the results on large number of instances, which may be rather confusing, only the results of GA only on the MLUFLP instances generated from a subset of basic ORLIB and M* instances are presented in this section.

Tables 5 and 6 summarize the GA result on these instances. In the first column instance names are given. The instance name carries the information about the initial instance, the number of levels, the number of facilities on each level and the number of clients. For example, the instance *capb_3l_12_25_63.1000* is created by modifying ORLIB instance *capb*, which has 3 levels with 12, 25, 63 facilities, respectively, and 1 000 clients.

The second column contains optimal solution on the current instance, if it is previously known, otherwise the sign – is written. The best GA value (GA_{best}) is given in the following column, with the mark "optimal" in cases when GA reached the optimal solution known in advance (obtained either by enumeration technique

Instance name	Opt_{sol}	GA_{best}	t sec	t_{tot} sec	gen	N_{best}	agap %	σ %	eval	cache %
cap71_1l_16.50	932 615.750 000	optimal	0.012	0.606	2 010.1	20	0.000	0.000	3 507.7	96.5
cap101_1l_25.50	796 648.437 500	optimal	0.030	0.781	2 026.3	20	0.000	0.000	9 614.3	90.5
cap131_1l_50.50	793 439.562 500	optimal	0.199	2.037	2 139.8	20	0.000	0.000	30 453.8	71.6
mq1_1l_300.300	3 591.273 000	optimal	14.288	63.289	2 306.7	20	0.000	0.000	68 927.1	40.3
mr1_1l_500.500	2 349.856 000	optimal	74.852	220.295	2 595.3	20	0.000	0.000	82 950.8	36.1
ms1_1l_1000.1000	4 378.632 000	best known	534.888	1 097.619	2 979.7	20	0.000	0.000	100 084.3	32.9
mt1_1l_2000.2000	9 176.509 000	best known	4 134.325	5 580.506	3 871.2	20	0.000	0.000	136 795.1	29.4
capa_1l_100.1000	17 156 454.478 300	optimal	13.409	77.864	2 319.3	20	0.000	0.000	52 072.8	55.1
capb_1l_100.1000	12 979 071.581 430	optimal	43.642	120.241	3 047.2	18	0.060	0.186	76 178.9	50.1
capc_1l_100.1000	11 505 594.328 780	optimal	34.542	102.535	2 907.3	12	0.073	0.135	70 952.4	51.2
cap71_2l_6_10.50	1 813 375.512 500	optimal	0.006	0.610	2 009.5	20	0.000	0.000	3 917.9	96.1
cap71_3l_2_5_9.50	4 703 216.306 250	optimal	0.005	0.557	2 010.4	20	0.000	0.000	4 300.5	95.7
cap101_2l_8_17.50	1 581 551.393 750	optimal	0.018	0.646	2 017.4	20	0.000	0.000	7 198.9	92.9
cap101_3l_3_7_15.50	3 227 179.812 500	optimal	0.019	0.612	2 018.3	20	0.000	0.000	6 797.8	93.3
cap131_2l_13_37.50	1 592 548.450 000	optimal	0.118	1.319	2 078.0	20	0.000	0.000	16 751.4	83.9
cap131_3l_6_14_30.50	3 201 970.462 500	optimal	0.105	1.276	2 108.2	6	0.125	0.084	19 877.3	81.3
cap131_4l_3_7_15_25.50	3 630 297.668 750	optimal	0.071	1.183	2 048.7	20	0.000	0.000	18 447.0	82.0

Table 5. GA results on the instances with previously known optimal/best solution

Instance name	Opt_{sol}	GA_{best}	t sec	t_{tot} sec	gen	N_{best}	agap %	σ %	eval	cache %
capa_2l_30_70.1000	-	31 524 957.410 085	8.380	49.235	2 308.2	5	0.106	0.063	50 829.1	56.0
capa_3l_15_30_55.1000	-	40 725 103.253 535	3.076	27.518	2 120.1	20	0.000	0.000	37 316.4	64.9
capa_4l_6_12_24_58.1000	-	54 643 362.801 190	4.688	36.480	2 194.9	10	0.881	0.904	46 738.3	57.5
capb_2l_35_65.1000	-	25 224 163.282 875	18.414	58.981	2 780.6	14	0.860	1.406	59 650.7	57.3
capb_3l_12_25_63.1000	-	34 978 486.506 140	3.137	23.788	2 089.6	20	0.000	0.000	25 763.8	75.4
capb_4l_6_13_31_50.1000	-	53 034 149.833 035	4.652	29.840	2 222.3	4	0.110	0.057	41 192.1	63.0
capc_2l_32_68.1000	-	22 762 468.837 500	22.625	63.857	3 020.8	8	0.770	0.717	63 538.1	58.1
capc_3l_13_27_60.1000	-	35 540 649.433 070	8.539	38.997	2 406.2	12	0.675	1.176	43 836.3	63.5
capc_4l_4_9_27_60.1000	-	57 017 358.038 275	4.601	39.784	2 166.0	13	0.150	0.210	44 774.9	58.7
mq1_2l_100_200.300	-	8 341.287 000	19.313	60.511	2 670.4	20	0.000	0.000	77 167.7	42.2
mq1_3l_30_80_190.300	-	12 994.871 000	16.980	55.666	2 616.2	3	2.273	1.369	75 305.6	42.4
mq1_4l_18_39_81_162.300	-	18 048.030 500	14.876	49.008	2 619.9	8	0.736	0.876	75 445.1	42.4
mq1_4l_20_40_80_160.300	-	17 648.009 500	17.423	51.654	2 698.7	11	1.764	2.124	77 449.5	42.6
mr1_2l_160_340.500	-	6 707.505 000	83.116	204.099	2 918.3	14	0.611	0.988	91 545.4	37.3
mr1_3l_55_120_325.500	-	10 911.319 000	76.009	187.238	2 858.1	2	1.341	0.814	89 511.4	37.4
mr1_4l_30_65_140_265.500	-	15 311.469 000	61.234	159.532	2 773.3	2	1.544	0.879	86 230.1	37.8
ms1_2l_320_680.1000	-	13 416.805 000	540.534	1 032.551	3 322.9	11	0.510	0.485	110 437.6	33.5
ms1_3n_120_250_630.1000	-	21 881.384 000	501.034	998.641	3 227.8	2	2.260	1.312	107 150.1	33.6
ms1_4l_64_128_256_552.1000	-	30 936.585 000	418.521	833.667	3 224.6	7	2.258	1.019	106 793.9	33.8
ms1_5l_25_55_120_250_550.1000	-	40 191.230 500	396.686	814.261	3 104.9	12	1.738	1.118	103 072.1	33.7
mt1_2l_650_1350.2000	-	27 733.057 000	3949.573	5 494.346	4 309.1	16	0.331	0.704	150 321.4	30.2
mt1_3l_256_600_1144.2000	-	46 095.090 000	3247.064	4 874.861	4 169.9	11	2.021	1.105	145 827.9	30.0
mt1_4l_120_250_520_1110.2000	-	65 044.002 500	3084.885	4 644.064	4 153.6	7	1.126	0.649	145 332.5	30.0
mt1_5l_60_120_250_500_1070.2000	-	83 523.753 000	2944.08	4 444.225	4 143.9	11	1.678	0.830	144 979.3	30.0

Table 6. GA results on the instances with unknown optimal solution

or CPLEX solver). If the GA reached the best-known solution for single-level cases, which is not proved to be optimal, the mark "best known" is written.

Average time needed to detect the best GA value is given in the t column, while t_{tot} represents the total running time (in seconds) needed for finishing GA. On average, GA finished after *gen* generations. The number of cases out of 20 when GA reaches GA_{best} solution is given in N_{best} column.

The solution quality in all 20 executions is evaluated as a percentage gap named *agap* with respect to the optimal solution Opt_{sol} or GA_{best}, with standard deviation σ of the average gap. They are defined as $agap = \frac{1}{20}\sum_{i=1}^{20} gap_i$, where $gap_i = 100 \times \frac{GA_i - Opt_{sol}}{Opt_{sol}}$ ($gap_i = 100 \times \frac{GA_i - GA_{best}}{GA_{best}}$ in cases when Opt_{sol} is not known) and GA_i represents the GA solution obtained in the i-th run, while σ is the standard deviation of gap_i, $i = 1, 2, \ldots, 20$, obtained by formula $\sigma = \sqrt{\frac{1}{20}\sum_{i=1}^{20}(gap_i - agap)^2}$. The last two columns are related to the caching: *eval* represents the average number of evaluations, while *cache* displays savings (in per cent) achieved by using caching technique.

It is evident from Table 5 that the proposed GA method reaches all previously known optimal solutions on ORLIB instances and M* instances for $k = 1$. For two large-scale instances, $ms1\_1n\_1000.1000$ and $mt1\_1n\_2000.2000$, no optimal solution is known up to now, but the GA reaches all best known solutions presented in [35]. For small size instances with multi-levels, the optimal solutions are obtained with total enumeration or CPLEX solver. The GA method is successful on these instances, reaching optimal solutions in short CPU time. The proposed method also provides solutions on large-scale M* and modified ORLIB instances in reasonable amount of CPU time.

In order to support the claim that duplicated individuals have to be removed, some experiments without removing duplicated individuals (GA_{dup}) are performed. For small size instance cap71_2l_6_10.50 the optimal value (equal to GA_{best}) is also reached by GA_{dup}. For instance capa_3l_15_30_55.1000 GA_{dup} obtained solution value 43400224.60963 (best in 20 runs), which has gap 6.16 % from GA_{best}. This gap is very large, so further testing of GA_{dup} seems to be useless.

GA concept cannot prove optimality and adequate finishing criteria that will fine-tune solution quality does not exist. Therefore, as column t_{tot} in Tables 5 and 6 shows, our algorithms run through additional $t_{tot} - t$ time (until finishing criteria is satisfied), although they already reached the optimal/best solution.

It can be seen from the column *cache*[%] in Tables 5 and 6 that significant percentage of run-time savings is achieved by using caching technique. On average, GA reused between 71.6 % and 96.5 % of the values from the cache-queue table while solving small size instances (up to 50 clients and up to 50 facilities). On larger data set, the percentage of savings was between 29.4 % and 75.4 %.

For large-scale modified ORLIB and M* instances, direct comparisons with existing methods for solving the MLUFLP can not be carried out. The MLRR, PR-RR and PR-LI algorithms use linear relaxation of the model with enormous number of variables and constrains. For example, for instance with $n = 2\,000$ clients and

$m = 2\,000$ facilities, these methods would involve 2×10^{15} variables and around $2\,000 \times 2\,000 = 4\,000\,000$ constraints. Since the integer linear programming problem is NP-hard, all solvers have exponential complexity, so the running time is $O(c^{2*10^{15}})$. Obviously, even for small c value in CPLEX and other solvers, it is impossible for these methods to obtain any solution in reasonable CPU time. The SP method is capable to solve larger size instances, but the gaps from optimal solutions are enormously large (up to 732 %). Search space for GA is obviously 2^m, since the binary encoding is applied in genetic algorithm with m genes. For previous instance, search space is large ($2^{2\,000} \approx 10^{600}$) but much smaller than search space for integer linear programming model (1)–(5). That is direct implication of using dynamic programming in objective function of the GA.

On the other hand, as the results in Tables 5 and 6 show, the GA easily works even for practical size instances. Since the proposed GA approach quickly reaches all previously known optimal/best solutions for single-level instances and for multi-level small size instances, it is reasonable to believe that GA also gave high-quality solutions on large-scale instances.

It can be seen From the t_{tot} column in Table 5 that the proposed GA quickly solves small size instances (up to $m = 50$, $n = 50$) to optimality. The maximal t_{tot} time of the GA on these instances was less than 2.04 seconds. The instances of the same size were used in [14], but in even these cases, the proposed SP method produced solutions with large gap. On larger-scale instances, the GA also gives solutions in reasonable CPU time (up to 5 580 seconds), as can be seen from Table 6.

6 CONCLUSIONS

In this paper, a robust evolutionary metaheuristics for solving the multi-level uncapacitated facility location problem is presented. Enormous number of feasible sequences of facilities in large-scale instances with more than two levels is significantly decreased by using dynamic programming approach with relatively small number of states. This approach helps the GA to use binary encoding and standard genetic operators to reach promising search regions. Computational experiments on generated instances demonstrate the robustness of the proposed algorithm with respect to both solutions' quality and running times even on large-scale MLUFLP instances.

Future research will be directed to parallelization of presented GA, hybridization with other heuristics and its application in solving similar facility location problems.

Acknowledgement

This research was partially supported by the Serbian Ministry of Science and Ecology under project 144007. The author is grateful to Jozef Kratica for his useful suggestions and comments and to Zorica Stanimirovic for useful comments on a draft version of this paper. The author also appreciates the constructive comments of the referees and the editor, resulting in an improved presentation.

REFERENCES

[1] Aardal, K.—Chudak, F.—Shmoys, D. B.: A 3-Approximation Algorithm for the K-Level Uncapacitated Facility Location Problem. Information Processing Letters, Vol. 72, 1999, pp. 161–167.

[2] Ageev, A.: Improved Approximation Algorithms for Multilevel Facility Location Problems. Operations Research Letters, Vol. 30, 2002, pp. 327–332.

[3] Ageev, A.—Ye, Y.—Zhang, J.: Improved Combinatorial Approximation Algorithms for the K-Level Facility Location Problem. SIAM Journal on Discrete Mathematics, Vol. 18, 2005, pp. 207–217.

[4] Beasley, J. E.: OR Library: Distributing Test Problems by Electronic Mail. Journal of the Operational Research Society, Vol. 41, 1990, pp. 1069–1072.

[5] Beasley, J. E.: Obtaining Test Problems via Internet. Journal of Global Optimization, Vol. 8, 1996, pp. 429–433.

[6] Bumb, A. F.—Kern, W.: A Simple Dual Ascent Algorithm for the Multilevel Facility Location Problem. Proceedings of the 4th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems and 5th International Workshop on Randomization and Approximation Techniques in Computer Science: Approximation, Randomization and Combinatorial Optimization, August 18–20, 2001, pp. 55–62.

[7] Charikar, M.—Guha, S.: Improved Combinatorial Algorithms for the Facility Location and K-Median Problems. In FOCS: IEEE Symposium on Foundations of Computer Science (FOCS), 1999.

[8] Chudak, F.—Shmoys, D.: Improved Approximation Algorithms for Capacitated Facility Location Problem. In Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, 1999, pp. 875–876.

[9] Correa, F. A.–Lorena, L. A. N.—Senne, E. L. F: Lagrangean Relaxation with Clusters for the Uncapacitated Facility Location Problem. Proceedings of the XIII Congreso Latino-Iberoamericano de Investigacion Operativa – CLAIO, Montevideo, Urugvay 2006.

[10] Djurić, B.—Kratica, J.—Točić, D.—Filipović, V.: Solving the Maximally Balanced Connected Partition Problem in Graphs by Using Genetic Algorithm. Computing and Informatics, Vol. 27, 2008, No. 3, pp. 341–354.

[11] Drezner, Z.—Hamacher, H.: Location Theory: Applications and Theory. Springer-Verlag, Berlin-Heidelberg 2002.

[12] Dias, J. M.—Caprtivo, M. E.—Clmaco, J.: A Memetic Algorithm for Dynamic Location Problems. MIC2005: Proceedings of the Sixth Metaheuristics International Conference, 2005, pp. 284–289.

[13] Dias, J. M.–Caprtivo, M. E.—Clmaco, J.: A Hybrid Algorithm for Dynamic Location Problems. Instituto de Engenharia de Sistemas e Computadores de Coimbra, working paper, No. 3, 2005.

[14] Edwards, N. J.: Approximation Algorithms for the Multi-Level Facility Location Problem. Ph. D. Thesis, Cornell University, 2001.

[15] Filipović, V.—Kratica, J.—Tošić, D.—Ljubić, I.: Fine Grained Tournament Selection for the Simple Plant Location Problem. Proceedings of the 5th OnlineWorld Conference on Soft Computing Methods in Industrial Applications – WSC5, September 2000, pp. 152–158.

[16] Filipović, V.: Fine-Grained Tournament Selection Operator in Genetic Algorithms. Computing and Informatics, Vol. 22, 2003, pp. 143–161.

[17] Filipović, V.: Operatori Selekcije I Migracije iWeb Servisi Kod Paralelnih Evolutivnih Algoritama. Ph. D. thesis, University of Belgrade, Faculty of Mathematics, 2006.

[18] Greistorfer, P.—Regoy, C.—Alidaee, B.: A Filter-and-Fan Approach for the Facility Location Problem. MIC2003: Proceedings of the Fifth Metaheuristics International Conference, 2003, pp. 27.1–27.6.

[19] Greistorfer, P.–Rego, C.: A Simple Filter-And-Fan Approach to the Facility Location Problem. Computers and Operations Research, Vol. 33, 2006, No. 9, pp. 2590–2601.

[20] Hansen, P.—Brimberg, J.—Urosević, D.—Mladenović, N.: Primal-Dual Variable Neighborhood Search for the Simple Plant-Location Problem. INFORMS Journal on Computing, Vol. 19, 2007, No. 4, pp. 552–564.

[21] Klose, A.—Drexl, A.: Facility Location Models for Distribution System Design. European Journal of Operational Research, Vol. 162, 2005, pp. 4–29.

[22] Kovacevic, J.: Hybrid Genetic Algorithm For Solving The Low-Autocorrelation Binary Sequence Problem. Yugoslav Journal of Operations Research (submitted for publication).

[23] Krarup, J.—Pruzan, P. M.: The Simple Plant Location Problem: Survey and Synthesis. European Journal of Operational Research, Vol. 12, 1983, pp. 36–81.

[24] Kratica, J.: Improving Performances of the Genetic Algorithm by Caching. Computers and Artificial Intelligence, Vol. 18, 1999, pp. 271–283.

[25] Kratica, J.: Improvement of Simple Genetic Algorithm for Solving the Uncapacitated Warehouse Location Problem. Advances in Soft Computing – Engineering Design and Manufacturing, R. Roy, T. Furuhashi and P. K. Chawdhry (Eds.), Springer-Verlag London Limited, 1999, pp. 390–402.

[26] Kratica, J.—Tošić, D.—Filipović, V.—Ljubić, I.: Solving the Simple Plant Location Problem by Genetic Algorithms. RAIRO – Operations Research, Vol. 35, 2001, pp. 127–142.

[27] Kratica, J.–Stanimirović, Z.–Točić, D.–Filipović, V.: Two Genetic Algorithms for Solving the Uncapacitated Single Allocation P-Hub Median Problem. European Journal of Operational Research Vol. 182, 2007, No. 1, pp. 15–28.

[28] Kratica, J.—Kovačević-Vujčić, V.—Cangalović, M.: Computing the Metric Dimension of Graphs by Genetic Algorithms. Computational Optimization and Applications, DOI 10.1007/s10589-007-9154-5 (to appear).

[29] Kratica, J.—Kovačević-Vujčić, V.—Cangalović, M.: Computing Strong Metric Dimension of Some Special Classes of Graphs by Genetic Algorithms. Yugoslav Journal of Operations Research (to appear).

[30] Ljubić, I.—Raidl, G. R.: A Memetic Algorithm for Minimum-Cost Vertex-Biconnectivity Augmentation of Graphs. Journal of Heuristics, Vol. 9, 2003, pp. 401–427.

[31] Ljubić, I.: Exact and Memetic Algorithms for Two Network Design Problems. Ph. D. thesis, Institute of Computer Graphics, Vienna University of Technology, 2004.

[32] Medaglia, A. L.–Gutiérrez, E.—Villegas, J. G.: A Tool for Rapid Development of Multi-Objective Evolutionary Algorithms (MOEAs) with Application to Facility Location Problems. MO-JGA: Implementing MOEAs in Java, 2005.

[33] Mitchell, M.: Introduction to Genetic Algorithms. MIT Press, Cambridge, Massachusetts 1999.

[34] Puchinger, J.—Raidl, G. R.—Pferschy, U.: The Core Concept for the Multi-Dimensional Knapsack Problem. Lecture Notes in Computer Science, Vol. 3906, 2006, pp. 195–208.

[35] Raidl, G. R.—Gottlieb, J.: Empirical Analysis of Locality, Heritability and Heuristic Bias in Evolutionary Algorithms: A Case Study for the Multidimensional Knapsack Problem. Evolutionary Computation, Vol. 13, 2006, No. 4, pp. 441–475.

[36] Resende, M. G. C.—Werneck, R. F.: A Hybrid Multistart Heuristic for the Uncapacitated Facility Location Problem. European Journal of Operational Research, Vol. 174, 2006, No. 1, pp. 54–68.

[37] ReVelle, C. S.—Eiselt, H. A.: Location Analysis: A Synthesis and Survey. European Journal of Operational Research, Vol. 165, 2005, pp. 119.

[38] Sevkli, M.—Guner, A. R.: A New Approach to Solve Uncapacitated Facility Location Problems by Particle Swarm Optimization. Proceedings of 5th International Symposium on Intelligent Manufacturing Systems, Sakarya University, Department of Industrial Engineering, May 29–31, 2006, pp. 237–246.

[39] Stanimirović, Z.—Kratica, J.—Dugošija, Dj.: Genetic Algorithms for Solving the Discrete Ordered Median Problem. European Journal of Operational Research, Vol. 182, No. 3, 2007, pp. 983–1001.

[40] Stanimirović, Z.: Genetic Algorithms for Solving Some NP-Hard Hub Location Problems. Ph. D. thesis, University of Belgrade, Faculty of Mathematics, 2007.

[41] Stanimirović, Z.: A Genetic Algorithm Approach for the Capacitated Single Allocation P-Hub Median Problem. Computing and Informatics, Vol. 29, 2010, No. 1, pp. 117–132.

[42] Sun, M.: Solving the Uncapacitated Facility Location Problem Using Tabu Search. Computers and Operations Research, Vol. 33, 2006, No. 9, pp. 2563–2589.

[43] Villegas, J. G.—Palacios, F.—Medaglia, A. L.: Solution Methods for the Bi-Objective (Cost-Coverage) Unconstrained Facility Location Problem With an Illustrative Example. Annals of Operations Research, Vol. 147, 2006, No. 1, pp. 109–141.

[44] Vygen, J.: Approximation Algorithms for Facility Location Problems (Lecture Notes). Research Institute for Discrete Mathematics, University of Bonn, Report No. 05950-OR, 2005.

[45] Yigit, V.—Turkbey, O.: An Approach to the Facility Location Problems With Hill-Climbing and Simulated Annealing. Journal of The Faculty of Engineering and Architecture of Gazi University, Vol. 18, 2003, No. 4, pp. 45–56.

[46] Zhang, J.: Approximating the Two-Level Facility Location Problem Via a Quasi-Greedy Approach. Mathematical Programming, Vol. 108, 2006, pp. 159–176.

Miroslav Marić received his B. Sc. degree in computer science (2002), M. Sc. in computer science (2006) and Ph. D. in computer science (2008) from the University of Belgrade, Faculty of Mathematics. Since 2009 he is Assistant Professor at the Faculty of Mathematics, University of Belgrade. His research interests include genetic algorithms, parallel algorithms and operational research, bioinformatics, computer graphics.