Annals of Operations Research 122, 21–42, 2003

An Efficient Genetic Algorithm for the p-Median Problem

OSMAN ALP and ERHAN ERKUT Erhan.Erkut@ualberta.ca
School of Business, University of Alberta, Edmonton, Alberta, T6G 2R6, Canada

ZVI DREZNER
Department of Management Science/Information Systems, California State University, Fullerton, CA 92634-6848, USA

Abstract. We propose a new genetic algorithm for a well-known facility location problem. The algorithm is relatively simple and it generates good solutions quickly. Evolution is facilitated by a greedy heuristic. Computational tests with a total of 80 problems from four different sources with 100 to 1,000 nodes indicate that the best solution generated by the algorithm is within 0.1% of the optimum for 85% of the problems. The coding effort and the computational effort required are minimal, making the algorithm a good choice for practical applications requiring quick solutions, or for upper-bound generation to speed up optimal algorithms.

Keywords: facility location, p-median, genetic algorithm, heuristic

1. Introduction

In this paper we propose a new genetic algorithm for the p-median problem, which is arguably the most popular model in the facility location literature. The goal of the model is to select the locations of p facilities to serve n demand points so as to minimize the total travel between the facilities and the demand points. This is a combinatorial optimization problem shown to be NP-hard by Cornuejols, Fisher and Nemhauser (1977). Many researchers devised heuristic methods to solve large instances of this problem to near optimality with reasonable computational effort.

Genetic algorithms (GAs) are heuristic search methods that are designed to mimic the evolution process. New solutions are produced from old solutions in ways that are reminiscent of the interaction of genes. GAs have been applied with success to problems with very complex objective functions. While a number of applications to combinatorial optimization problems have been reported in the literature, there are few applications of genetic algorithms to facility location problems. The GA we describe in this paper has a number of desirable properties: it is simple, it generates excellent solutions, and it is fast.

In section 2, we introduce the p-median problem formally and briefly describe the relevant literature. In section 3, we discuss the properties of GAs in general terms and review the cross-section of the p-median and GA literatures. We describe our GA in

section 4, and provide a small numerical example in section 5. Section 6 contains the results of our computational experience with the GA.

2. The p-median problem

The p-median model is a location/allocation model, which locates p facilities among n demand points and allocates the demand points to the facilities. The objective is to minimize the total demand-weighted distance between the demand points and the facilities. The following formulation of the p-median problem is due to ReVelle and Swain (1970).

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} w_i d_{ij} x_{ij} \\
\text{s.t.} \quad & \sum_{j=1}^{n} x_{ij} = 1 && \forall i, \\
& x_{ij} \leqslant y_j && \forall i, j, \\
& \sum_{j=1}^{n} y_j = p, \\
& x_{ij} = 0 \text{ or } 1 && \forall i, j, \\
& y_j = 0 \text{ or } 1 && \forall j,
\end{aligned}
$$

where

$$
\begin{aligned}
n &= \text{total number of demand points}, \\
x_{ij} &= \begin{cases} 1 & \text{if point } i \text{ is assigned to facility located at point } j, \\ 0 & \text{otherwise}, \end{cases} \\
y_j &= \begin{cases} 1 & \text{if a facility is located at point } j, \\ 0 & \text{otherwise}, \end{cases} \\
w_i &= \text{demand at point } i, \\
d_{ij} &= \text{travel distance between points } i \text{ and } j, \\
p &= \text{number of facilities to be located.}
\end{aligned}
$$

This is an uncapacitated facility location model where every demand point is served by one facility and trips to demand points are not combined. It is a useful strategic planning tool for many single-level distribution systems (for application examples see (Fitzsimmons and Austin, 1983; Erkut, Myroon and Strangway, 2000)).

Teitz and Bart (1968) presented one of the oldest and most popular heuristic algorithms for this problem. It is essentially an exchange heuristic that starts with a random solution and improves it iteratively by swapping facilities in and out of the solution. If one uses multiple starts, this method generates very good solutions when applied to smaller problems. Mathematical programming-based heuristics were suggested by

Narula, Ogbu and Samuelsson (1977), which combines Lagrangian relaxation with subgradient optimization, and Galvão (1980), which utilizes the dual of the problem. More recently, Densham and Rushton (1992) proposed a two-phase search heuristic, and Pizzolato (1994) suggested a decomposition heuristic that works on a forest of p trees.

Modern heuristics have been applied to the p-median problem as well. A Tabu search algorithm was designed by Rolland, Schilling and Current (1997). Simulated annealing algorithms were designed by Murray and Church (1996) and Chiyoshi and Galvão (2000). Rosing, ReVelle and Schilling (1999) proposed a composite heuristic which found optimal solutions with regularity. We discuss GAs for the p-median problem in the next section.

Despite the combinatorial nature of the problem, optimal algorithms can solve some instances of the p-median problem with reasonable effort. A typical approach is to combine Lagrangian relaxation with branch-and-bound (see, for example, (Daskin, 1995)). Galvão and Raggi (1989) complemented this approach with a primal–dual algorithm. Similarly, Koerkel (1989) combined a primal–dual algorithm with branch-and-bound. In contrast, Avella and Sassano (2001) presented two classes of inequalities and use them in a cutting plane algorithm.

3. Genetic algorithms

GAs are computational solution procedures that have been inspired by biological progression. They are intensive search heuristics that allow solutions to evolve iteratively into good ones. They were first proposed as problem-solving methods in the 1960s, but they have become popular in the operations research literature more recently. Thorough treatments of GAs can be found in (Reeves, 1993; Dowsland, 1996).

The chromosomes in a GA correspond to solutions in an optimization problem. The one-to-one relationship between a chromosome and the corresponding solution is determined by an appropriate encoding. There is a fitness function that evaluates the quality of a chromosome. Pairs of chromosomes are selected by the fitness function and crossed to produce new chromosomes – this is the primary mechanism by which solutions evolve. Mutations are used to promote genetic diversity (hence to facilitate a thorough search of the feasible region). GAs work well for complex optimization problems since they preserve the common sections of the chromosomes that have high fitness values. They consistently disregard poor solutions and evaluate more and more of the better solutions.

Although the general principles of GAs are common to all GA applications, there is no generic GA, and the user has to custom-design the algorithm for the problem at hand. This is not a trivial task since a GA requires many design decisions. The encoding is critical since a poor choice may result in a poor algorithm regardless of its other features. Other important decisions include the size of the population in one generation (i.e., the number of solutions), the selection of parents (selection of two old solutions to produce new solutions), the crossover operator (the method by which new solutions are

produced by old solutions), the replacement of one generation by the next, the method and frequency of the mutations, and the number of generations.

In principle, GAs can be applied to any optimization problem. Given that GAs have been applied to many combinatorial optimization problems with success (Reeves, 1993), we expect them to work well on location/allocation problems. Yet little effort has been directed towards designing GAs for location problems in general, and for the p-median problem in particular. In the first paper applying the GA framework to the p-median problem, Hosage and Goodchild (1986) encoded the solutions of the problem as a string of n binary digits (genes). This encoding does not guarantee the selection of exactly p facilities in each solution, and the authors use a penalty function to impose this constraint. Primarily due to this encoding choice the algorithm performs rather poorly even on very small problems.

Dibble and Densham (1993) describe another GA for a multi-criteria facility location problem, solving a problem with $n = 150$ and $p = 9$ with a population size of 1000 and 150 generations. They report that their algorithm finds solutions that are almost as good as the solutions generated by an exchange algorithm, but with more computational effort. Moreno-Perez, Moreno-Vega and Mladenovic (1994) design a parallelized GA for the p-median problem, where multiple population groups exist and individuals are exchanged between these groups. This feature has effects similar to mutation since it prevents premature homogenization in the population groups. They do not report computational results.

In the most recent application of GA to the p-median problem, Bozkaya, Zhang and Erkut (2002) describe a fairly complex GA and report the results of an extensive computational experiment with algorithm parameters. This algorithm is able to produce solutions that are better than the solutions of an exchange algorithm. However, convergence is very slow. In contrast, the algorithm described in the next section is much simpler and produces good solutions very quickly.

4. A new genetic algorithm

4.1. Encoding

We use a simple encoding where the genes of a chromosome correspond to the indices of the selected facilities. For example, $(5, 7, 2, 12)$ is a chromosome that corresponds to a feasible solution for a 4-median problem where demand points 2, 5, 7, and 12 are selected as facility locations. This encoding ensures that the last constraint in the formulation is always satisfied.

4.2. Fitness function

The fitness function is the objective function of the p-median problem. The fitness of a chromosome is identical to the objective function value of the solution it corresponds to, and it can be calculated easily using the problem data. The calculations assume

that every demand point would be allocated to one facility, namely to the closest open facility. This ensures that the first two constraints in the formulation are always satisfied. Hence, the selections of the fitness function and the encoding satisfy all constraints and no additional effort is needed in the implementation of the algorithm to enforce the constraint set.

4.3. Population size

The GA works on a population with fixed size. Large populations slow down the GA while small populations may not have sufficient genetic diversity to allow for a thorough search over the feasible region. Hence, the selection of the population size is important.

We target population sizes with the following two properties:

1. Every gene must be present in the initial population. Since the GA iterates by creating new combinations of the genes in the original gene pool, an incomplete gene pool will result in a partial search over the feasible region, and the GA would have to rely on mutations to bring missing genes into the pool. The minimum number of members to represent each gene in the initial population is $\lceil n/p \rceil$, the smallest integer greater-than-or-equal to n/p.

2. The population size should be proportional to the number of solutions. In general, the larger the feasible region of a problem, the more difficult it is to find the best solution.

Our goal is to devise a formula to generate population sizes with these properties for different problem parameters. Let $S = C\binom{n}{p}$ be the number of all possible solutions to the problem, and $d = \lceil n/p \rceil$ the rounded-up density of the problem. The population size we suggest for a particular problem is

$$P(n, p) = \max\left\{2, \left\lceil \frac{n}{100} \cdot \frac{\ln(S)}{d} \right\rceil\right\} d.$$

According to this formula, the population size is an integer multiple of d, and this satisfies the first property. In fact, the result of the max operator is at least two, guaranteeing that every gene appears at least twice in the initial population. (If each gene appears only once in the initial population, it might disappear – once and for all – in an early iteration which may make it difficult to find optimal solutions.) The $\ln(S)$ term provides the increase in the population size in proportion to S, the number of solutions. Yet the increase is very slow (due to the ln operator), which keeps the population size manageable even for very large problems such as $n = 1000$ and $p = 100$. Figures 1–3 show how this population formula size behaves as a function of n and p.

There are many other formulas that have the two properties listed above. The one we suggest is merely one that works. It generates very good solutions over a large spectrum of problem parameters in our empirical tests. It may be possible to find a formula that generates better solutions or works faster – we do not claim that our formula is "optimal" in some sense.

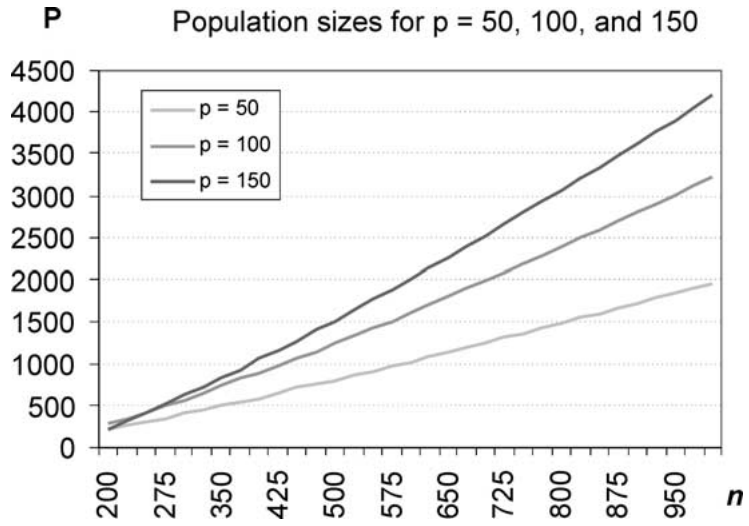

Figure 1. Population size as a function of n for three different values of p.

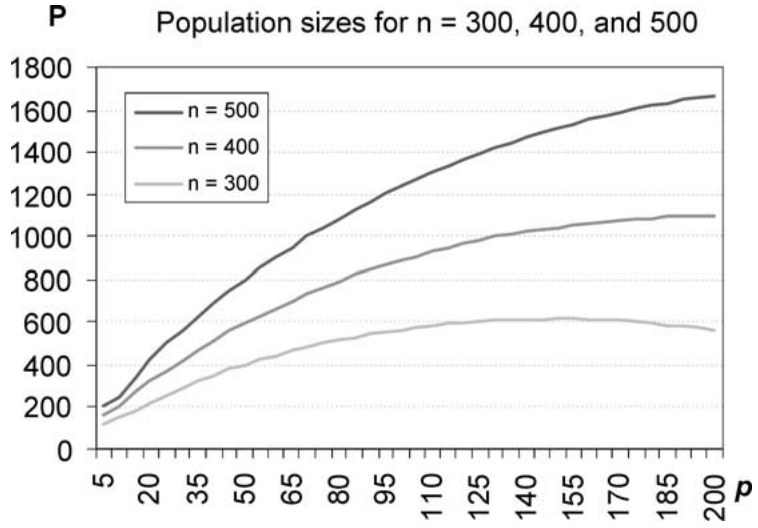

Figure 2. Population size as a function of p for three different values of n.

4.4. Initializing the population

As discussed above, every gene should be present in the initial pool. It is also desirable to have each gene approximately with the same frequency in the initial pool so the gene pool is not biased. Suppose that the population size is equal to kd, for some constant k. For the first set of n/p members, we assign the genes $1, 2, \ldots, p$ to the first member, the genes $p+1, p+2, \ldots, 2p$ to the second member, and so on. For the second set of n/p members, we distribute the genes similarly, but we use an increment of two in

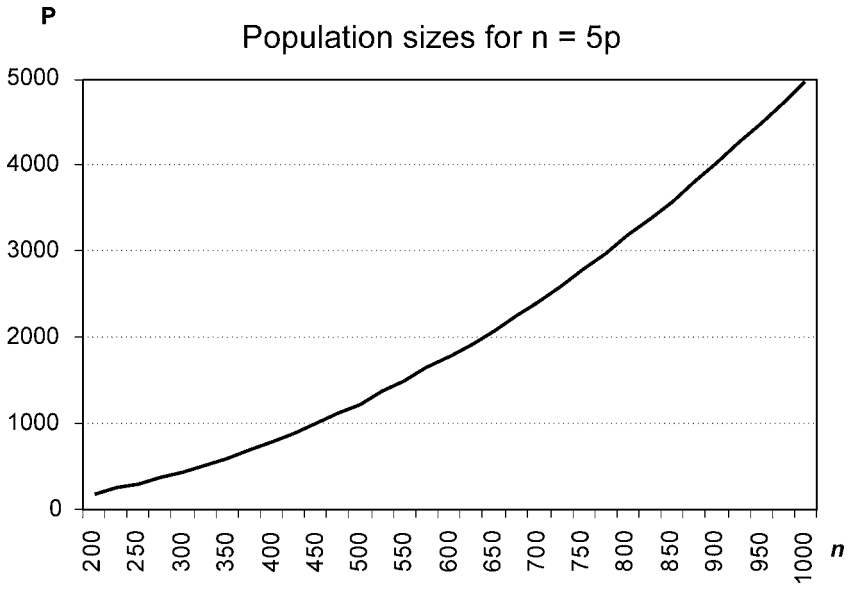

Figure 3. Population size as a function of n for problems with $n = 5p$.

the sequences. For example, we assign the genes $1, 3, 5, \ldots, 2p-1$ to the first member in the second group. Similarly, in the kth group we distribute the genes to the members sequentially with an increment of k. For example, for $(n, p, k) = (12, 4, 2)$, we would have the following initial population: $(1, 2, 3, 4)$, $(5, 6, 7, 8)$, $(9, 10, 11, 12)$, $(1, 3, 5, 7)$, $(9, 11, 2, 4)$, $(6, 8, 10, 12)$. If n/p is an integer then each gene is represented in the initial population with an equal frequency. If n/p is not an integer then after distributing all of the genes from 1 to n to each group, we allocate random genes to fill empty slots.

4.5. Selecting the parents

The parents are selected randomly from the population. We experimented with biased selection mechanisms where fitter parents are more likely to be selected for reproduction, but experienced no significant impact on the performance of the algorithm. While both versions of the algorithm produced excellent results, we obtained marginally better results with randomly selected parents.

4.6. Generating new members

In a typical GA two parents are selected for reproduction, and their chromosomes are merged in a prescribed way to produce two children. Usually the chromosomes of the parents are split into two, creating four partial chromosomes, and then these four pieces are combined to create two new chromosomes. For example, the parents $(1, 2, 3, 4, 5)$ and $(6, 7, 8, 9, 10)$ would create the children $(1, 2, 3, 9, 10)$ and $(6, 7, 8, 4, 5)$ if a crossover after the third gene is used. We do not use such a traditional crossover operator. Instead, we take the union of the genes of the parents, obtaining an infeasible

solution with m genes where $m > p$. Then, we apply a greedy deletion heuristic to decrease the number of genes in this solution one at a time until we reach p. However, we never drop genes that are present in both parents. To reduce the number of genes by one we discard the gene whose discarding produces the best fitness function value (i.e., increases the fitness value by the least amount). We call the infeasible solution obtained after the union operation the "*draft member*" and the feasible solution generated by the heuristic as the "*candidate member*".

The generation operator can be summarized as follows.

Generation Operator.
Input: Two different members.

Step 1. Take the union of the input members' genes to obtain a draft member.

Step 2. Let the total number of genes in this draft member be m. Call the genes that are present in both parents *fixed genes* and the rest *free genes*.

Step 3. Compute the fitness value of this draft member.

Step 4. Find the *free gene* that produces the minimum increase in the current fitness value when deleted from the draft member, and delete it. Repeat this step until $m = p$. Let this final solution be a candidate member.

Output: A candidate member.

Our children generation method is distinctly different from a crossover operator. We use a greedy heuristic instead of blindly creating children by cut-and-paste of chromosome parts. This increases time demands, but it also improves the quality of the children. While this can be viewed as compromising a major strength of the GA, our computational experience shows that this operator works well. Berman, Drezner and Wesolowsky (2002) report success with a similar merge-drop operator in solving another location problem.

4.7. Mutation

The mutation operator is an important component of a GA. This operator helps the algorithm to escape from local optima. Although we experimented with different mutation techniques (for example, in 2% of new member generation iterations we added a facility not found in the union of the two parents before starting the generation operator, and did not allow the deletion of this facility), they did not improve the performance of the algorithm. Hence we decided not to use the mutation operator.

4.8. Replacement

We admit a candidate member into the population, if it is distinct (i.e., not identical to an existing member), and if its fitness value is better than the worst fitness value in the

population, by discarding the worst member. Such a policy improves the average fitness value of the population gradually while maintaining genetic diversity. We store the worst and best members of the population after every population update.

The steps for the replacement operator are as follows.

Replacement Operator.
Input: One candidate member.

Step 1. If fitness value of the input candidate member is higher than the maximum fitness value in the population, then discard this candidate member and terminate this operator.

Step 2. If the candidate member is identical to an existing member of the current population, then discard this candidate member and terminate this operator.

Step 3. Replace the worst member of the population with the input candidate member.

Step 4. Update the worst member of the population.

Step 5. Update the best member of the population.

Output: Population after introducing the candidate member.

4.9. Termination

After experimenting with different iteration counts that depend on the problem parameters, we opted for a convergence-based stopping criterion. The algorithm terminates after observing $\lceil n\sqrt{p} \rceil$ successive iterations where the best solution found has not changed. One iteration consists of one use of the generation and replacement operators. For example, for a problem with $(n, p) = (100, 20)$, the algorithm terminates if 448 successive children fail in improving the best solution. (For all of our test problems, we had $n > 2p$. For a problem with $n \leqslant 2p$, we would suggest stopping after $n\sqrt{(n - p)}$ non-improving iterations.) An alternative termination rule (stop after a given number of successive iterations where the *entire population* has not changed) generated better solutions, but required more computational effort.

4.10. Algorithm

The overall algorithm can be stated as follows.

Algorithm.

1. Generate an initial population of size $P(n, p)$ as described in section 4.4.
2. Initialize a variable for keeping track of successive iterations where the best solution found has not changed, *MaxIter*.
3. Repeat while *MaxIter* $\leqslant \lceil n\sqrt{p} \rceil$:

3.1 Randomly select two members from the current population.

3.2 Run the Generation Operator: input these two members and obtain a candidate member.

3.3 Run the Replacement Operator: input candidate member if possible.

3.4 If the best solution found so far has not changed, then increment *MaxIter*.

4. Select the best member as the final solution of the algorithm.

5. Numerical example

We use a very small problem with $n = 12$ and $p = 3$ to provide a numerical example. The coordinates of the 12 demand points are given in table 1, and the Euclidean metric is used to measure distances. For $P = 8$ the initial population along with their fitness values is shown in table 2.

In the first iteration, members 1 and 4 are combined. The resulting draft member is 1-2-3-10-11-12 with a fitness value of 136. Three genes have to be dropped from this member to produce a feasible solution. Dropping 12 results in the smallest increase in the fitness value (to 156). Then we drop 2 and 1, and generate the candidate member 3-10-11 with fitness value 241. This is not a current member of the population and its fitness value is lower than that of the worst. Hence, we delete 2-4-6 and introduce 3-10-11 as member 7 of the population in its place.

In iteration 2, members 4 and 7 are selected for merger. The resulting draft member is 3-10-11-12 with a fitness value of 210. Genes 10 and 11 are fixed since they appear in both parents. The greedy drop heuristic selects gene 12 for deletion, and the resulting

Table 1
The coordinates of the 12 demand points (unweighted).

Point	1	2	3	4	5	6	7	8	9	10	11	12
x	2	2	29	22	2	67	98	33	25	95	36	31
y	55	91	91	99	70	99	52	6	44	88	36	71

Table 2
The initial population.

No.	Member	Fitness
1	1-2-3	352
2	4-5-6	316
3	7-8-9	348
4	10-11-12	257 (best)
5	1-3-5	358
6	7-9-11	365
7	2-4-6	391 (worst)
8	8-10-12	271

Table 3
Summary of the first 10 iterations of GA.

Iter.	Merge	Draft	Candidate	Fitness	Replace	Comment
1	1 and 4	1-2-3-10-11-12	3-10-11	241	7	new best
2	4 and 7	3-10-11-12	3-10-11	241	–	identical to 7
3	8 and 2	4-5-6-8-10-12	5-8-10	266	6	
4	1 and 2	1-2-3-4-5-6	1-3-6	282	5	
5	2 and 7	3-4-5-6-10-11	3-10-11	241	–	identical to 7
6	3 and 5	1-3-6-7-8-9	3-7-9	245	1	
7	1 and 8	3-7-8-9-10-12	3-9-10	236	3	new best
8	4 and 5	1-3-6-10-11-12	3-10-11	241	–	identical to 7
9	2 and 6	4-5-6-8-10	5-8-10	266	–	identical to 6
10	1 and 2	3-4-5-6-7-9	2-6-9	262	2	

member is 3-10-12. This is identical to the current member 7, and the population remains unchanged. In iteration 3, members 2 and 8 are merged to create 4-5-6-8-10-12. From this draft, the greedy heuristic generates candidate member 5-8-10 with fitness value 266, which replaces member 6 in table 2. The first 10 iterations are summarized in table 3. The optimal solution is 3-9-10, which is found in iteration 7.

6. Computational study

6.1. Accuracy and speed of the GA

To test the performance of our algorithm (abbreviated as ADE here), we solved four sets of test problems:

1. *OR Library:* There are 40 p-median problems with known optimal solutions in the OR Library (Beasley, 1990). The problem sizes range from $n = 100$ to 900 and $p = 5$ to 200.

2. *Alberta:* We generated a 316-node network using all population centers in Alberta. We computed distances using shortest paths on the actual road network and used populations for weights. We solved 11 different problems on this network with values of p ranging from 5 to 100, and we found the optimal solutions using Daskin's (1995) SITATION software.

3. *Galvão:* We solved 16 random problems from (Galvão and ReVelle, 1996) with $n = 100$ and $n = 150$ and various values of p.

4. *Koerkel:* We solved 13 problems from (Koerkel, 1989) for p values ranging from 2 to 333 on a network with 1,000 nodes.

(The Alberta, Galvão, and Koerkel problem sets are available through the Internet: www.bus.ualberta.ca/eerkut/testproblems)

In sum, we solved 80 problems of different sizes (ranging from 100 to 1,000 nodes) from different sources, including random as well as realistic problems. The computational results for 10 replications of ADE, produced by a C++ code on a Pentium III 733 MHz computer with 128 MB memory, are summarized in table 4. In 28 of the 40 problems, ADE finds an optimal solution. The average deviation between the best (worst) solution value found in 10 replications and the optimal objective function value is 0.036% (0.291%). Another measure of ADE's performance is the average deviation between the *average* of the best solution values and the optimal values in 10 replications: 0.11%. In other words, the average of the best solutions over the 10 replications is 100.11% of the optimal. These near-optimal solutions are generated with little computational effort. ADE is very fast with an average per-replication time of 18.4 seconds for the 40 test problems. The per-replication times range from 0.1 seconds for $(n, p) = (100, 5)$ to 2.2 minutes for $(n, p) = (900, 90)$.

Table 5 summarizes the results for the other three sets of test problems. All Alberta problems and approximately half of the Galvão and Koerkel problems are solved optimally. The average gap between the best solution found and the optimum is 0.05% for the Galvão problems and 0.10% for the Koerkel problems. The per-replication time spent on the Alberta and Galvão problems are 3.2 seconds and 0.5 seconds, respectively. The 1,000-node Koerkel problems take longer: an average of 3 minutes per replication. These times are similar to those spent on OR Library problems of comparable sizes.

Based on our experience with the 80 test problems, it is fair to say that ADE is quite fast and find excellent solutions. The best solution found is optimal for more than half of the problems. The average gap between the best solution and the optimum for all the 80 problems is 0.045%, and the worst gap is 0.4%. Most of the replications take under a minute and the longest one takes 3 minutes.

6.2. Comparison with other heuristics

In this section we first compare ADE using the OR Library problems with three other heuristics: the genetic algorithm (BZE) by Bozkaya, Zhang and Erkut (2002), the Gamma heuristic (Gamma) suggested by Rosing, ReVelle and Schilling (1999), the simulated annealing (SA) algorithm by Chiyoshi and Galvão (2000). In addition we compare ADE with the three heuristics in the SITATION code (myopic, exchange, neighbourhood) using the Alberta and Galvão problem sets. (The 1,000-node Koerkel problems are too large for the SITATION code.)

Although BZE and ADE are both GAs, there are a number of differences between them. BZE uses the same encoding as ADE. Its initial population is generated at random while ADE uses a systematic way to seed the initial population. BZE favors parents with better fitness values whereas ADE selects parents at random. BZE uses three different crossover operators while ADE uses a merge-drop heuristic to generate a child. BZE uses mutations and invasions (a more intense form of mutation) to maintain genetic diversity, but ADE has no such features. BZE replaces the entire population with the next generation while keeping the best solution in the population. ADE introduces children

Table 4
Summary of the performance of the algorithm on the OR Library problems.

Problem	n	p	Optimal	Best % deviation[a]	Worst % deviation[b]	Average % deviation[c]	Duration (sec.)[d]
1	100	5	5,819		0.84	0.14	0.1
2	100	10	4,093		1.54	0.20	0.1
3	100	10	4,250		1.36	0.33	0.2
4	100	20	3,034		0.53	0.29	0.2
5	100	33	1,355				0.3
6	200	5	7,824		0.28	0.03	0.4
7	200	10	5,631		0.55	0.21	0.5
8	200	20	4,445				0.7
9	200	40	2,734		0.48	0.30	1.2
10	200	67	1,255	0.08	0.56	0.29	2.0
11	300	5	7,696		0.38	0.16	1.7
12	300	10	6,634		0.27	0.03	1.2
13	300	30	4,374				2.1
14	300	60	2,968		0.03	0.01	4.4
15	300	100	1,729	0.23	0.46	0.39	6.3
16	400	5	8,162		0.26	0.03	2.3
17	400	10	6,999		0.06	0.01	2.4
18	400	40	4,809		0.04	0.00	5.6
19	400	80	2,845	0.04	0.21	0.12	13.3
20	400	133	1,789	0.17	0.28	0.22	16.3
21	500	5	9,138		0.43	0.09	3.8
22	500	10	8,579				4.5
23	500	50	4,619		0.09	0.05	15.9
24	500	100	2,961	0.03	0.14	0.10	21.1
25	500	167	1,828	0.22	0.33	0.27	31.6
26	600	5	9,917		0.07	0.01	6.8
27	600	10	8,307		0.18	0.04	7.8
28	600	60	4,498	0.02	0.04	0.02	24.5
29	600	120	3,033	0.07	0.23	0.13	43.7
30	600	200	1,989	0.40	0.60	0.48	79.0
31	700	5	10,086		0.19	0.03	14.5
32	700	10	9,297		0.15	0.02	13.2
33	700	70	4,700		0.09	0.04	45.4
34	700	140	3,013	0.07	0.10	0.09	65.2
35	800	5	10,400		0.26	0.03	15.6
36	800	10	9,934				18.5
37	800	80	5,057	0.02	0.08	0.05	75.9
38	900	5	11,060		0.41	0.09	28.8
39	900	10	9,423				26.5
40	900	90	5,128	0.10	0.12	0.10	132.2

[a] The percentage deviation of the best solution found in 10 replications from the optimal solution.
[b] The percentage deviation of the worst solution found in 10 replications from the optimal solution.
[c] The percentage deviation of the average of the 10 solutions from the optimal solution.
[d] The average per-replication duration of the algorithm in seconds.

Table 5
Summary of the performance of the algorithm on the Alberta, Galvão, and Koerkel problems.

Problem	No.	n	p	Optimal	Best % deviation[a]	Worst % deviation[b]	Average % deviation[c]	Duration (sec.)[d]
Alberta	1	316	5	82173923		9.77	1.23	1.4
Alberta	2	316	10	42140049		0.54	0.11	1.5
Alberta	3	316	20	24401799				2.1
Alberta	4	316	30	16196223				2.5
Alberta	5	316	40	11123076				2.9
Alberta	6	316	50	8191619				3.3
Alberta	7	316	60	6251379				3.7
Alberta	8	316	70	4683041				4.1
Alberta	9	316	80	3498764				4.4
Alberta	10	316	90	2669093				4.7
Alberta	11	316	100	2058746				5.1
Galvão	1	100	5	5703				0.1
Galvão	2	100	10	4426	0.32	2.76	1.11	0.1
Galvão	3	100	15	3893		0.23	0.10	0.2
Galvão	4	100	20	3565		0.25	0.12	0.2
Galvão	5	100	25	3291	0.03	0.15	0.09	0.2
Galvão	6	100	30	3032		0.07	0.02	0.2
Galvão	7	100	35	2784		0.18	0.06	0.3
Galvão	8	100	40	2542		0.12	0.06	0.3
Galvão	9	150	5	10839		0.62	0.08	0.2
Galvão	10	150	15	7390		0.77	0.45	0.5
Galvão	11	150	20	6454	0.12	0.67	0.31	0.6
Galvão	12	150	25	5875		0.70	0.23	0.6
Galvão	13	150	35	5192	0.04	0.37	0.23	0.9
Galvão	14	150	45	4636	0.11	0.26	0.19	0.9
Galvão	15	150	50	4374	0.14	0.21	0.18	0.9
Galvão	16	150	60	3873	0.08	0.21	0.13	1.2
Koerkel	1	1000	2	46118255		0.27	0.09	53.6
Koerkel	2	1000	4	32110068		0.09	0.06	77.2
Koerkel	3	1000	6	26007551		0.17	0.06	67.0
Koerkel	4	1000	8	22251618		0.08	0.01	71.8
Koerkel	5	1000	10	19706508	0.00	0.12	0.04	85.8
Koerkel	6	1000	12	17804044		0.06	0.02	80.8
Koerkel	7	1000	17	14785148		0.03	0.01	98.2
Koerkel	8	1000	25	12004788	0.04	0.21	0.09	175.0
Koerkel	9	1000	50	8036540	0.02	0.18	0.11	184.1
Koerkel	10	1000	111	4810938	0.22	0.38	0.25	348.0
Koerkel	11	1000	143	4019654	0.17	0.30	0.25	315.1
Koerkel	12	1000	200	3117365	0.41	0.50	0.46	372.2
Koerkel	13	1000	333	1923368	0.37	0.41	0.40	444.3

[a] The percentage deviation of the best solution found in 10 replications from the optimal solution.
[b] The percentage deviation of the worst solution found in 10 replications from the optimal solution.
[c] The percentage deviation of the average of the 10 solutions from the optimal solution.
[d] The average per-replication duration of the algorithm in seconds.

Table 6
A summary of the comparison of the four heuristics on the 40 OR Library problems.

	BZE	ADE	SA	Gamma
Number of problems solved optimally	10	28	26	39
Average deviation from optimum (%)	2.174	0.036	0.083	0.001
Average per-replication time (sec.)	4430.6	18.4	90.7	1093.4

into the population as they are produced as long as they improve the average fitness of the population. BZE is tested on small problems – the largest one being $(n, p) = (100, 10)$. For these problems, population sizes of 50, 75, and 100 are used and up to 100,000 new solutions are generated. In contrast, for $(n, p) = (100, 10)$ ADE uses a population size of 40 and generates 450 solutions on average.

The algorithm proposed by Chiyoshi and Galvão (2000) uses vertex exchanges, similar to the Teitz and Bart algorithm, to generate neighborhood solutions and uses them within the context of simulated annealing. The authors test the performance of their algorithm on the OR Library problems. In contrast, the Gamma heuristic is a three-stage heuristic based on the heuristic concentration principles outlined by Rosing and ReVelle (1997). In the first stage, the Teitz and Bart algorithm is applied a number of times, and a subset consisting of the best solutions are retained. The facilities that are selected in all of the retained solutions are fixed in stage 2, which applies a 2-opt heuristic to select the remaining facilities. The third stage applies a 1-opt heuristic to the final solution of stage 2 allowing for every facility to be swapped out of the solution.

Table 6 contains a summary of the comparison of the solutions generated by these four heuristics on the 40 OR Library problems, as well as the computational times. When comparing the two GAs, we note that ADE finds solutions that are closer to the optimum than BZE in less time. On average, the ADE solution is 0.04% above the optimum while the BZE solution is 2.17% above. BZE can find optimal solutions only for problems with small values of p, and its performance seems to deteriorate as p increases. ADE can find optimal or near-optimal (within 0.1% of the optimum) solutions in all but 4 problems. The ratio of the average time used up BZE over ADE is 240 – ADE is faster than BZE by two orders of magnitude on a comparable machine. It seems that on the 40 test problems used, ADE is clearly a better algorithm than BZE.

The SA's performance is closer to ADE's. Yet ADE finds better solutions in less time. ADE's average gap is half SA's, and ADE's average time is four times faster than SA's. The Gamma heuristic is very effective on the 40 test problems used. It finds the optimal solution in 39 of the 40 problems. The average deviation between the best (worst) fitness value found in 5 replications and the optimal objective function value is 0.001% (0.065%). The average deviation between the best solutions in 5 replications and the optimal values is 0.024%. All of these figures are better than those for ADE. However, both heuristics find excellent solutions and the differences are very small. Time comparison between ADE and the Gamma heuristic is not straightforward since the two heuristics were run on different machines. The Gamma heuristic was executed on a Sun Sparc server 250 with 800 MB of free core which has a slower processor but more memory

Table 7
A summary of the comparison of the ADE against the SITATION heuristics (as well as the optimal solution found by SITATION using Lagrangian relaxation and branch and bound) using the 11 Alberta problems and the 16 Galvão problems.

Alberta (11 problems)	ADE	Myopic	Exchange	Neighb.	Optimal
Number of problems solved optimally	11	0	7	3	11
Average deviation from optimum (%)	0.00	4.49	0.11	0.25	0.00
Average per-replication time (sec.)	3.25	–[a]	–[a]	–[a]	6.1
Galvão (16 problems)	**ADE**	**Myopic**	**Exchange**	**Neighb.**	**Optimal**
Number of problems solved optimally	9	2	2	2	16
Average deviation from optimum (%)	0.05	1.17	0.65	1.76	0.00
Average per-replication time (sec.)	0.5	–[a]	–[a]	–[a]	2824

[a] These times are not reported in SITATION.

than our machine. Keeping this difference in mind, the average ADE computational time is an order of magnitude shorter than Gamma's. It seems that Gamma finds slightly better solutions but consumes considerably more computational effort than ADE.

Table 7 summarizes the comparison of ADE against the three start-up heuristics used in SITATION on Alberta and Galvão problems. For both sets, ADE's solutions are considerably better than those of the SITATION heuristics. Perhaps the most noteworthy aspect of this comparison is the computational effort required by SITATION to find optimal solutions to Galvão problems. While 10 of the 16 problems are solved to optimality in less than one minute, some take several hours. Problem 11 with $(n, p) = (150, 20)$ took over 8 hours to solve to optimality. On that problem, the best of the three SITATION heuristic solutions was 2.45% away from the optimum. In contrast ADE's solution, found in 6 seconds with 10 replications, was only 0.12% away. SITATION might be able to solve this problem faster if it is provided with the sharper upper bound generated by ADE.

Based on our limited comparison with three other recently published heuristics and three heuristics coded in SITATION, we conclude that our algorithm is competitive. The only heuristic that generates better solutions than our algorithm spends considerable more time. It seems that our algorithm could be used with success in cases where quick-and-dirty solutions are required for large practical problems, or to generate good starting solutions for optimal algorithms.

6.3. Characteristics of the algorithm

In the first two parts of this section we established that our algorithm is fast, finds near-optimal solutions with regularity, and it competes well against better heuristics. In this section we explore the characteristics of the algorithm to better understand how it works and why it works well. For this part, we focus on problem #15 from the OR Library with $(n, p) = (300, 100)$, a problem that was not solved optimally by our algorithm.

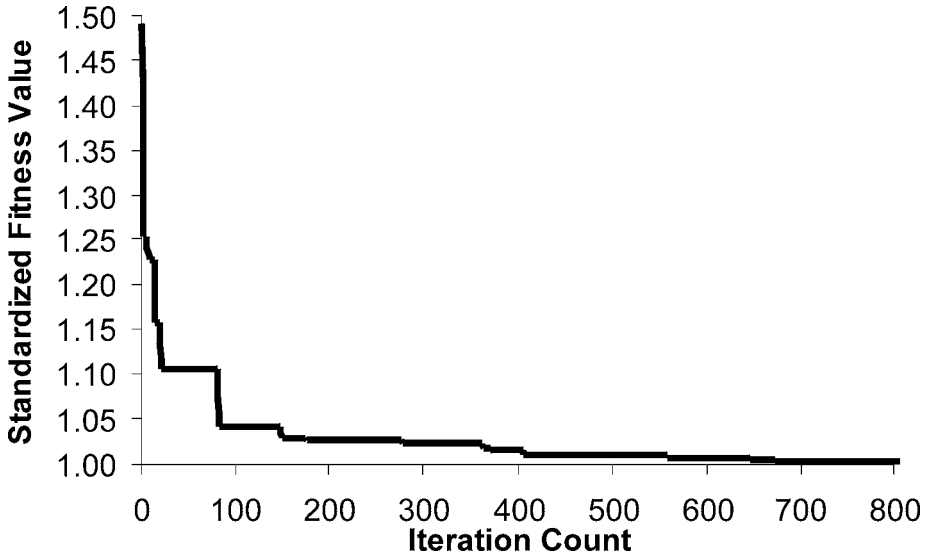

Figure 4. The improvement of the best fitness value as a function of the iteration count for OR Library problem #15.

A desirable feature of the algorithm is that it finds very good solutions rather quickly. As an example, the improvement of the best fitness value for problem #15 over 800 iterations is displayed in figure 4. For this problem, although the GA performs a total of 6,326 iterations, it finds a solution that is within 11% of the optimum after only 19 iterations (i.e., offspring), and a solution that is within 1% of the optimum after 559 iterations. The last fitness improvement occurs at iteration 3,327. This sharp drop in the fitness values over iterations is in stark contract with the behavior of GAs (for example, see (Bozkaya, Zhang and Erkut, 2002), for a similar figure), and it is quite similar to behavior of greedy heuristics.

The population size for this problem is 564. Figure 5 shows the average population fitness value over iterations. This figure indicates that the average fitness value decreases linearly during the first 1,000 iterations, and changes very little after that. Figure 6 shows the variance of the fitness value of the population. The variance increases significantly over the first 300 or so iterations as solutions that are considerably better than the current solutions are brought into the population. After 300 iterations the variance starts falling just as sharply as it increased, as the algorithm increases the concentration of good solutions in the population, dropping poor solutions. After 1,000 iterations, the variance drops to a small fraction of the initial variance (under 4%), and it keeps decreasing further to almost zero.

Perhaps figure 7 provides better insight into what happens to the population over time, and explains how the mean and variance change. The objective function values of the solutions are displayed on the x-axis. The optimal solution to this problem is 1,729 (near the origin). The histogram of the fitness values of the initial population is displayed using a light gray column chart (with a bin size of 50). The initial population

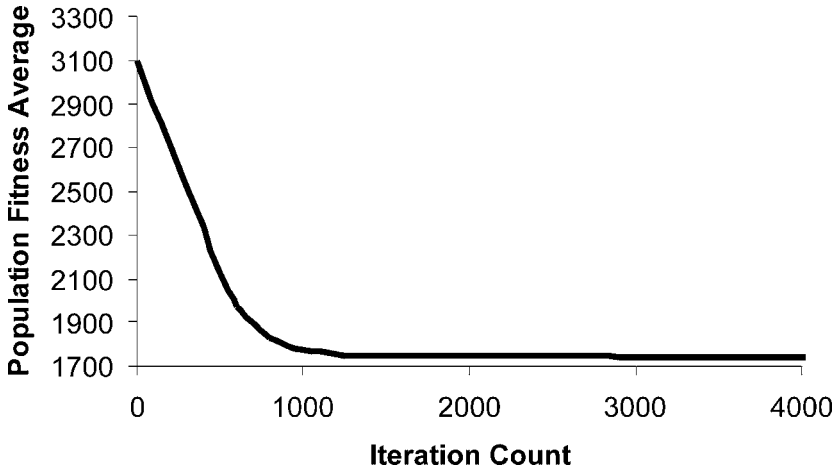

Figure 5. The reduction in the average population fitness value as a function of the iteration count for OR Library problem #15.

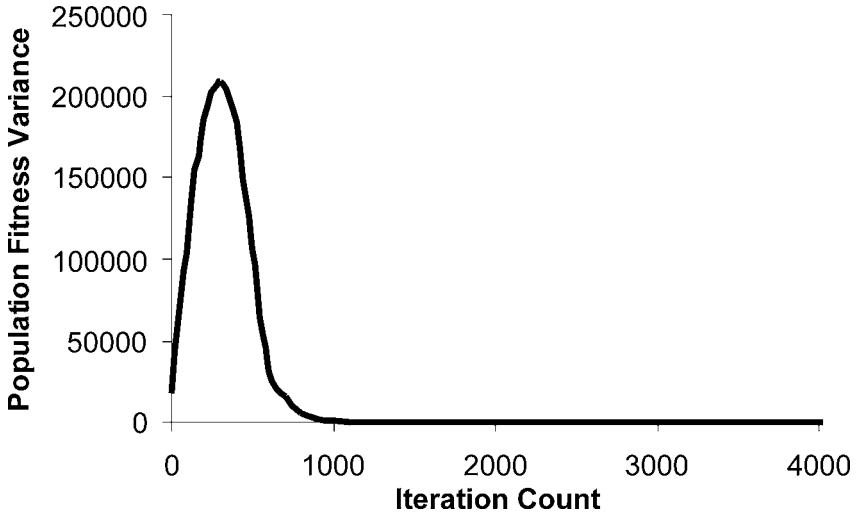

Figure 6. The variance of the population fitness values as a function of the iteration count for OR Library problem #15.

is merely a random sample of the set of entire solutions. The histograms of the fitness values after 200, 400, and 600 iterations are displayed using successively darker line charts. The line chart after 200 iterations shows that a large number of poor solutions have been deleted, and many good solutions have been introduced. This is a snapshot of the phase where the algorithm increases the variance of the fitness values by generating good solutions – note how the lightest line chart stretches towards the origin. After 400 iterations, more than half of the members of the initial population have been dropped, and a significant number of good solutions exist in the solution. After 600 iterations,

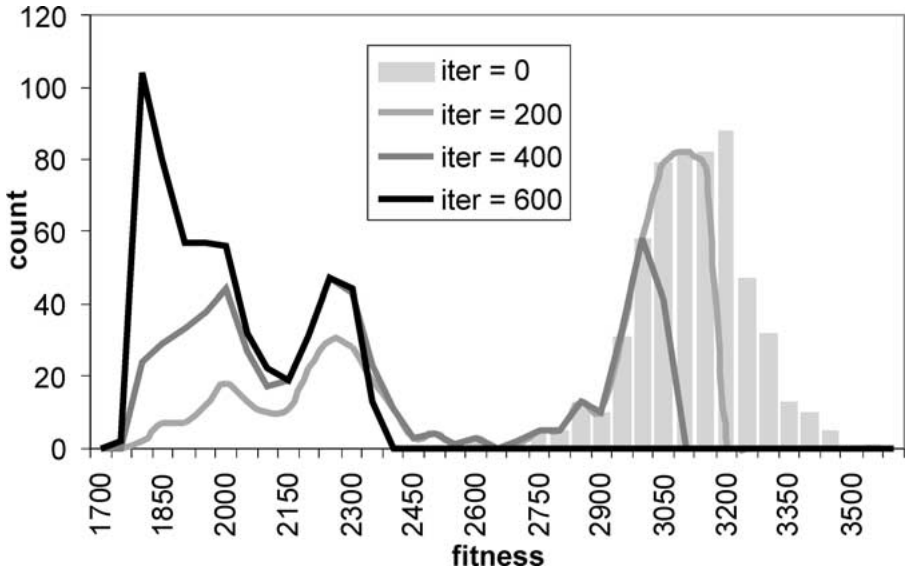

Figure 7. The histogram of the fitness values in the population for OR Library problem #15.

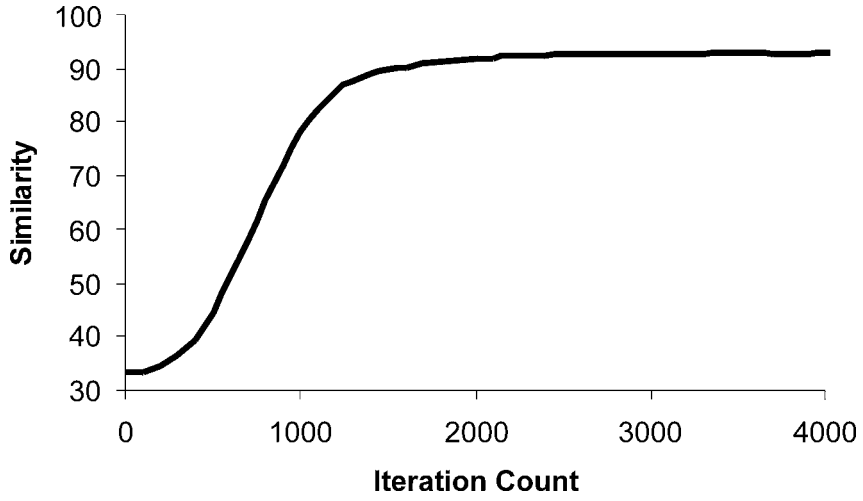

Figure 8. The similarity index of the population as a function of the iteration count for OR Library problem #15.

the entire initial population has been eliminated. While the distribution is bimodal, near-optimal solutions are gaining quickly over all others. We cannot show the histograms of further iterations on the same chart effectively since the distribution becomes extremely narrow and spikes near the optimum in latter iterations. When the algorithm stops, the best fitness value is 1,735 and the average fitness value is 1,742.4. All 564 members would be in the same bin in our histogram resulting in a single spike of length 564 for the range 1700–1750.

We believe figure 7 provides a good summary of the mechanics of the algorithm: it takes a normal distribution of fitness values, stretches it out to include good solutions, then produces increasingly higher concentrations of near-optimal solutions, dropping poor solutions, and reducing the variance drastically.

Figure 8 provides another picture of the variance reduction in the population by focusing on the similarity between the members. The similarity index between two members is defined as the percentage overlap of their genes. The similarity index of the population is the average of the individual similarity indices. Given the way we generate the initial population, the starting similarity index is 33.3. As figure 8 indicates, the similarity index over iterations is an S-shaped curve. It increases first slowly (as the good solutions are first introduced), then linearly, and starts leveling off after iteration 1,000, converging to a value of 92.4. While it is not surprising that near-optimal solutions are quite similar to one another, note that the similarity is not extreme. On average, 8 of the 100 locations are different between a pair of solutions at termination. Hence, we have a large number of very good solutions to the problem, and this may be more useful than one optimal solution.

7. Concluding remarks

In this paper we propose a new genetic algorithm for the p-median problem. The algorithm evolves solutions by taking the union of two solutions and dropping facilities one-at-a-time to generate a feasible solution. It is simple to code, and it generates near-optimal solutions very quickly. In 85% of the test problems it generated solutions that were within 0.1% of the optimum and its worst solution was only 0.41% away from the optimum. It is also quite fast. For example, a 500-node 50-median problem is solved in less than 16 seconds. We believe that this algorithm can be useful in instances where very good solutions are needed very quickly.

This algorithm does not use some of the features common in other genetic algorithms (such as mutation), and the operator used to generate new solutions is a greedy selection heuristic as opposed to a crossover operator. Hence, it may be more accurate to call it a *hybrid evolutionary heuristic* as opposed to a genetic algorithm. We experimented with nonrandom parent selection rules as well as mutation and invasion, and did not observe a marked improvement in the algorithm performance.

We believe that the greedy selection heuristic is responsible for the performance of the heuristic. Given any two parents, it generates one of the best possible offspring from the genes at hand. (In the context of genetics, we can think of the genes that are dropped by the greedy heuristic as recessive genes, and of those maintained as dominant genes. The analogy is not perfect since in our case a gene's classification as dominant or recessive would depend on the other genes of the two parents.) This results in the generation of a number of good solutions quickly. While a standard crossover operator merely crosses chromosomes with no attention paid to the quality of the resulting offspring, our operator can be thought of as an evolutionary offspring generator. Perhaps this is why it is not necessary to select parents according to fitness values. Even when

working with two average parents, the offspring generator can find "the best in them", and generate an offspring that is considerably better than both. Other features of the algorithm support the offspring generator. Every gene is present the same (or almost the same) number of times in the initial population, and premature homogenization of the population is prevented by not allowing duplications of chromosomes in the population. Perhaps these features negate the need for mutations and invasions in the algorithm.

While the greedy merge-drop heuristic generates near-optimal solutions quickly, it may also be responsible for the inability of the algorithm to generate optimal solutions more frequently. Manipulations of the solution generator, such as the replacement of the greedy selection rule by a semi-greedy selection rule, could improve the performance on some problems by slowing down convergence.

We made no effort to customize the algorithm to the problems on hand, and used the same formulas for the population size and the stopping criterion for all 80 problems we solved. It may be possible to improve the quality of the solutions generated, or to reduce the computational effort required, for any given problem by fine-tuning these two parameters, or by using a slight variation of the algorithm (such as a new termination rule, or a mutation operator). However, we find the performance of the simple version we experimented with quite satisfactory.

Acknowledgments

This research was supported by Natural Sciences and Engineering Research Council of Canada (OGP 25481). The authors are indebted to Burcin Bozkaya and Ken Rosing for their assistance with the computational experiment, to Roberto Galvão and Manfred Koerkel for providing test problems with known optimal solutions, and to Mark Daskin for providing a solution code.

References

Avella, P. and A. Sassano. (2001). "On the p-Median Polytope." *Mathematical Programming* 89(3), 395–411.

Beasley, J.E. (1990). "OR-Library – Distributing Test Problems by Electronic Mail." *Journal of the Operational Research Society* 41(11), 1069–1072.

Berman, O., Z. Drezner, and G.O. Wesolowsky. (2002). "Locating Unreliable Service Facilities that Are Distance Sensitive." *Computers and Operations Research*, in press.

Bozkaya, B., J. Zhang, and E. Erkut. (2002). "A Genetic Algorithm for the p-Median Problem." In Z. Drezner and H. Hamacher (eds.), *Facility Location: Applications and Theory*. Berlin: Springer.

Chiyoshi, F. and R.D. Galvão. (2000). "A Statistical Analysis of Simulated Annealing Applied to the p-Median Problem." *Annals of Operations Research* 96, 61–74.

Cornuejols, G., M.L. Fisher, and G.L. Nemhauser. (1977). "Location of Bank Accounts to Optimise Float: an Analytic Study of Exact and Approximate Algorithms." *Management Science* 23, 789–810.

Daskin, M.S. (1995). *Network and Discrete Location: Models, Algorithms and Applications*. New York: Wiley (SITATION can be downloaded from `http://users.iems.nwu.edu/~msdaskin/BookSoftware.htm` – as of July 2002).

Densham, P.J. and G. Rushton. (1992). "A More Efficient Heuristic for Solving Large p-Median Problems." *Papers in Regional Science* 71, 307–329.

Dibble, C. and P.J. Densham. (1993). "Generating Interesting Alternatives in GIS and SDSS Using Genetic Algorithms." In *GIS/LIS 1993*.

Dowsland, K.A. (1996). "Genetic Algorithms – A Tool for OR?" *Journal of Operational Research Society* 47, 550–561.

Erkut, E., T. Myroon, and K. Strangway. (2000). "TransAlta Redesigns Its Service Delivery Network." *Interfaces* 30(2), 54–69.

Fitzsimmons, J.A. and A.L. Austin. (1983). "A Warehouse Location Model Helps Texas Comptroller Select Out-of-State Audit Offices." *Interfaces* 13(5), 40–46.

Galvão, R.D. (1980). "A Dual-Bounded Algorithm for the p-Median Problem." *Operations Research* 28(5), 1112–1121.

Galvão, R.D. and L.A. Raggi. (1989). "A Method for Solving to Optimality Uncapacitated Location Problems." *Annals of Operations Research* 18, 225–244.

Galvão, R.D. and C. ReVelle. (1996). "A Lagrangean Heuristic for the Maximal Covering Location Problem." *European Journal of Operations Research* 88, 114–123.

Hosage, C.M. and M.F. Goodchild. (1986). "Discrete Space Location–Allocation Solutions from Genetic Algorithms." *Annals of Operations Research* 6, 35–46.

Koerkel, M. (1989). "On the Exact Solution of Large-Scale Simple Plant Location Problems." *European Journal of Operations Research* 39, 157–173.

Moreno-Perez, J.A., J.M. Moreno-Vega, and N. Mladenovic. (1994). "Tabu Search and Simulated Annealing in p-Median Problem." In *Proceedings of the Canadian Operational Research Society Conference*, Montreal.

Murray, A.T. and R.L. Church. (1996). "Applying Simulated Annealing to Location-Planning Models." *Journal of Heuristics* 2, 31–53.

Narula, S.C., U.I. Ogbu, and H.M. Samuelsson. (1997). "An Algorithm for the p-Median Problem." *Operations Research* 25, 709–712.

Pizzolato, N.D. (1994). "A Heuristic for Large-Size p-Median Location Problems with Application to School Location." *Annals of Operations Research*, 50 473–485.

Reeves, C.R. (1993). "Genetic Algorithms." In C.R. Reeves (ed.), *Modern Heuristic Techniques for Combinatorial Problems*. Chapter 4, pp. 151–196.

ReVelle, C. and R. Swain. (1970). "Central Facilities Location." *Geographical Analysis* 2, 30–42.

Rolland, E., D.A. Schilling, and J.R. Current. (1997). "An Efficient Tabu Search Procedure for the p-Median Problem." *European Journal of Operational Research* 96, 329–342.

Rosing, K.E. and C.S. ReVelle. (1997). "Heuristic Concentration: Two-Stage Solution Construction." *European Journal of Operational Research* 97, 75–86.

Rosing, K.E., C.S. ReVelle, and D.A. Schilling. (1999). "A Gamma Heuristic for the p-Median Problem." *European Journal of Operational Research* 117, 522–532.

Teitz, M.B. and P. Bart. (1968). "Heuristic Methods for Estimating the Generalized Vertex Median of a Weighted Graph." *Operations Research* 16, 955–961.