Pergamon

0305-0548(94)E0004-Q

Computers Ops Res. Vol. 21, No. 10, pp. 1103–1113, 1994

0305-0548/94 $7.00+0.00

A COMPARISON OF p-DISPERSION HEURISTICS

ERHAN ERKUT,[1,2]† YILMAZ ÜLKÜSAL[2]‡ and OKTAY YENIÇERIOĞLU[2]§

[1]Department of Finance and Management Science, Faculty of Business, University of Alberta, Edmonton, Alberta, Canada T6G 2R6 and [2]Department of Industrial Engineering, Boğaziçi University, Bebek, Istanbul 80815, Turkey

(Received November 1992; in revised form October 1993)

Scope and Purpose—Given a set of candidate points, we consider the problem of selecting a subset of points that are spread out as much as possible. This problem may be relevant for locating facilities that should not be clustered in one location, but be dispersed; such as franchises belonging to a chain. It also applies in the selection of a subset of efficient solutions for presentation to a decision-maker, if the set of all efficient solutions is too large for consideration. This problem is difficult to solve optimally. A number of quick-and-dirty solution procedures have been developed for this problem. The objective of this article is to compare 10 of these solution procedures. Our computational tests indicate that most of these procedures generate very good solutions in a few seconds on a microcomputer. It is possible to find excellent solutions by combining several of these solution procedures.

Abstract—The objective of the p-dispersion problem is to choose p out of n given points, such that the minimum distance between any pair of chosen points is as large as possible. Possible application areas include location theory and multicriteria optimization. The p-dispersion problem is known to be NP-hard. In this paper, we examine 10 heuristic methods for solving this problem, and provide a comparison of them based on several criteria. We report our computational experience with the heuristics on randomly generated planar problems of different sizes. Most of the heuristics generate very good solutions with very little computational effort on a microcomputer. We suggest performing multiple applications of several heuristics to minimize the possibility of finding poor solutions.

1. INTRODUCTION

In this paper, we present the results of an empirical comparison of solutions obtained by heuristic algorithms for the discrete p-dispersion problem. In the discrete p-disperion problem, we select p out of n given points ($1<p<n$) in some space, where the objective is to maximize the minimum distance between any two of the selected points. Defining V to be the set of n points $\{x_i;\ 1\leqslant i\leqslant n\}$, S to be a subset of V with $|S|=p$, and $\|\ \|$ to be a distance metric, we can express the p-dispersion problem as:

$$\max_{S\subseteq V} f(S)=\left[\min_{x_i\neq x_j\in S}\|x_i-x_j\|\right].$$

In the context of location theory, the maximin objective of the p-dispersion problem may be appropriate if closeness of the selected locations is not desirable. An example is the location of mutually obnoxious facilities, such as nuclear power plants, or missile silos. In these cases, closeness is undesirable due to the possibility of chain accidents. Another example is the location of franchises, where the maximal separation of franchises reduces intrachain competition. These and other examples are discussed in Refs [1, 2].

†Erhan Erkut is a Professor of Management Science at the University of Alberta. He also regularly teaches at the Industrial Engineering Department of Boğaziçi University. His main research interests are in the area of hazardous materials logistics. He has published articles in *Location Science, Transportation Science, Operations Research, European Journal of Operational Research, Interfaces, Networks*, and in other journals.

‡Yılmaz Ülküsal is an industrial engineer with Vestel in Constanta, Romania. He received a B.S. in Industrial Engineering from Boğaziçi University in 1992.

§Oktay Yenişerioğlu is a graduate student in Boğaziçi University. He received a B.S. in Industrial Engineering from Boğaziçi University in 1992.

Another application of the p-dispersion problem is in interactive multicriteria decision-making. The problem under consideration may have a very large number of efficient solutions. In this case, the analyst provides the decision-maker with a subset of all efficient solutions, and the decision-maker selects the most desirable solution. The analyst generates, and provides the decision-maker with, another set of efficient solutions in the neighborhood of the selected solution, and the process continues until the decision-maker is satisfied with an efficient solution. In this process, the selection of the set of efficient solutions to be presented to the decision-maker at each iteration is very important. The selected set should represent the entire set, and it should consist of solutions that are dissimilar. Steuer [3] recommends the use of the p-dispersion model to accomplish these objectives.

Although it is a fundamental problem in combinatorial optimization, the p-dispersion problem has not received much attention in the operations research literature until recently. In the oldest reference we can find, Steuer and Harris [4] treat this problem in the context of generating efficient points for a multicriteria problem. In a textbook on multiple criteria optimization, Steuer [3] describes several heuristics for this problem. Kuby [1] treats the p-dispersion problem in a locational context, and formulates the problem as an integer program. Wang and Kuo [5] develop an efficient algorithm for the one-dimensional version of the problem. More recently, Erkut [2] describes branch-and-bound methods and heuristics for this problem. Erkut and Neuman [6] compare solutions for the p-dispersion problem to the solutions of related problems. White [7, 8] and Ravi *et al.* [9, 10] treat the problem from a computational complexity perspective. Kincaid [11] proposes simulated annealing and tabu search heuristics for this problem.

The p-dispersion problem is known to be NP-hard [2, 5]. Thus, to solve large instances of the problem, one may have to rely on a heuristic. Furthermore, heuristics are useful in generating good solutions quickly, and providing upper bounds for an implicit enumeration algorithm. A number of heuristics for this problem have been proposed in the literature. Ravi *et al.* [9, 10] show that no polynomial time algorithm can guarantee a solution which is less than twice the optimum for the p-dispersion problem, for any distance metric (unless P = NP). This result implies that the worst-case solution of heuristics may not be a good criterion for choosing one, since worst-case solutions may be quite poor. Furthermore, computational experience with some of the heuristics implies that the average of their solutions is much better than their worst-case solution [2]. Thus, the average of the solutions of the heuristics may be a more suitable criterion for comparison. The goal of this paper is to provide this empirical comparison of heuristics for the p-dispersion problem. In the next section, we describe the heuristics we compare. In Section 3 we summarize the results of our computational experience. We present our conclusions and recommendations in Section 4.

2. HEURISTICS

The 10 heuristics we consider in this paper can be classified as follows:

- 2.1. Construction algorithms
  - 2.1.1. Greedy construction heuristic (GC)
  - 2.1.2. Greedy deletion heuristic (GD)
  - 2.1.3. Semi-greedy deletion heuristic (SG)
- 2.2. Neighborhood algorithms
  - 2.2.1. First point outside the neighborhood heuristic (FP)
  - 2.2.2. Closest point outside the neighborhood heuristic (CS)
  - 2.2.3. Furthest point outside the neighborhood heuristic (FS)
- 2.3. Projection algorithm (PR)
- 2.4. Interchange algorithms
  - 2.4.1. First pairwise interchange heuristic (IF)
  - 2.4.2. Best pairwise interchange heuristic (IM)
  - 2.4.3. Simulated annealing heuristic (SA)

For each heuristic, we now offer a brief description of it, discuss the related literature, and give a pseudo-code for it as well.

2.1. Construction algorithms

A solution to the p-dispersion problem consists of two sets of points. In one set, there are the p selected points, and in the other set there are the remaining n-p points. The construction algorithms process the set of given points, and change the status of one point at each iteration. Either a point is added to a solution set, or a point is deleted from it.

2.1.1. Greedy construction heuristic (GC). This heuristic is initialized by choosing the two furthest points in V. This is the optimal solution to the two-dispersion problem. Subsequently, in each of the next $(p-2)$ iterations, a new point is added to the solution. The point is chosen to maximize the minimum distance to the points already in the solution set. Every time a new point is added to the solution, the objective function decreases (assuming no distance is duplicated). The logic in this heuristic is to minimize the reduction in the objective function value at each iteration. This heuristic is used by Erkut and Neuman [6] to find solutions to the p-dispersion problem. It is also discussed by Rabi *et al.* [9], where its relative worst-case solution is established as twice the optimum. Figure 1 contains a pseudo-code of this heuristic using the same terminology as in Section 1. We also define the distance between a point and a set to be the smallest of the distances between the point and the members of the set. Namely, $d(x_k, S)=\min\{d(x_k, x_j): x_j \in S, x_j \neq x_k\}$:

1. Find x_1 and x_2 in V, s.t. $d(x_1, x_2)=\max\{d(x_i, x_j): 1 \leqslant i < j \leqslant n\}$
2. $S \leftarrow \{x_1, x_2\}$
3. While $|S| < p$ do
   3.1. Find $x_k \in V \backslash S$, s.t. $d(x_k, S)=\max\{d(x_j, S): x_j \in V \backslash S\}$
   3.2. $S \leftarrow S \cup \{x_k\}$

Fig. 1. Pseudo-code for the greedy construction heuristic (GC).

2.1.2. Greedy deletion heuristic (GD). This heuristic starts with a solution that contains all points, and eliminates one point at each iteration, until p points are left. The point to be eliminated is one of the two closest points in the current solution. Among those two points, the one that is closest to the remaining points in the solution is eliminated. The objective is to increase the objective function at each iteration, by eliminating one of the two points that define the objective function value. The rule for choosing between the two points is geared towards minimizing the potential for the kept point to define the objective function value in a future iteration. Figure 2 contains a pseudo-code of this heuristic.

1. $S \leftarrow V$
2. While $|S| > p$ do
   2.1. Find x_1, x_2, with $x_1 \neq x_2$ and $d(x_1, x_2)=\min\{d(x_i, x_j): x_i, x_j \in S\}$
   2.2. Find $x_k \in \{x_1, x_2\}$, s.t. $d(x_k, S)=\min\{d(x_j, S): j=1, 2\}$
   2.3. $S \leftarrow S \backslash \{x_k\}$

Fig. 2. Pseudo-code for the greedy deletion heuristic (GD).

2.1.3. Semi-greedy deletion heuristic (SG). This algorithm is a variant of the greedy deletion heuristic. The only difference is that the selection of which of the two closest points in the current solution should be eliminated is made randomly. Since no time is spent for choosing one of the two points, this heuristic is faster than its greedy counterpart. Another advantage is that one can generate several solutions to the problem through multiple applications of the heuristic. Since the p-dispersion problem is known to have many optimal solutions [6], this increases the chances of finding an optimal solution. Erkut [2] proposed this heuristic, and reports encouraging computational results (Fig. 3).

1. $S \leftarrow V$
2. While $|S| > p$ do
   2.1. Find x_1, x_2, with $x_1 \neq x_2$ and $d(x_1, x_2)=\min\{d(x_i, x_j): x_i, x_j \in S\}$
   2.2. Set $x_k = x_1$ or $x_k = x_2$ randomly
   2.3. $S \leftarrow S \backslash \{x_k\}$

Fig. 3. Pseudo-code for the semi-greedy deletion heuristic (SG)

This heuristic is "arbitrarily bad", implying that its worst case solution is as bad as possible [7]. We can demonstrate this with a simple example on the line. Suppose we have $(n+1)$ points on a line, where the distance between each pair of consecutive points is unity. Clearly, the optimal solution to the two-dispersion problem consists of points 1 and $(n+1)$ with a separation of n units. The semi-greedy deletion algorithm could start with points 1 and 2 and discard point 1. It could proceed by choosing points 2 and 3, discarding point 2. Continuing in this fashion, it could delete all points but points n and $(n+1)$, and report these two points as the solution to the problem. The distance between two points is one, being the worst solution to the problem. Note that the ratio of the objective function value of this solution to the optimal objective function value is $1/n$, and this ratio goes to zero as n goes to infinity. Hence the term "arbitrarily bad".

2.2. *Neighborhood algorithms*

Neighborhood algorithms for this problem have been proposed by Steuer and Harris [4], and Steuer [3]. They can be classified as construction heuristics, since they start with one point in the solution set, and add one point at each iteration. However, they all use the concept of the r-neighborhood of a point x_i, which is defined as $N(x_i, r)$,

$$N(x_i, r) = \{x \in R^n: \|x - x_i\| \leqslant r\}.$$

In R^2, the r-neighborhood of point x_i consists of a circle centered at point x_i with radius r. Given a radius r, and a point x_i, the neighborhood algorithms eliminate all points that are in the r-neighborhood of x_i from consideration. This guarantees that the minimum separation between the points in a solution generated using a neighborhood algorithm is at least r.

The neighborhood algorithms start with a point, which is called a "seed" by Steuer. At each subsequent iteration, one point outside the neighborhoods of the points in the solution is added to the solution. The algorithm stops when no candidate points are left. Clearly, the solution set is a function of r, namely $S(r)$. For a given value of r it is possible for this algorithm to terminate with $|S(r)|$ not equal to p. In fact, there may not be a value of r for which $|S(r)| = p$. In that case, one may have to adjust the solution set using greedy addition or deletion steps.

It is not obvious which value of r will result in $|S(r)|$ being closest to p. Thus, one may have to try different values of r. Due to the discrete nature of the problem, we can restrict the search over r to values in the distance matrix. This observation suggests that a binary search over a sorted distance array would be an efficient way of finding the value of r that results in $|S(r)|$ being closest to p. Figure 4 presents a general pseudo-code for all three neighborhood algorithms. The difference between these algorithms is in the selection rule of the point to be included in the solution set at step 3.1. Note that the neighborhood algorithms must be nested in a binary search over r, which is not shown in the pseudo-code.

The names of the three neighborhood algorithms specify the selection rule used in step 3.1 of the pseudo-code in Fig. 4. In the "first point outside the neighborhood" heuristic (FP), the point to be selected is the next point down the list of points. As the sequencing of the points in V may bias the solution, this heuristic should be run several times with point indices shuffled. In the "closest point outside the neighborhood" heuristic (CS), the point to be selected is the one that has the smallest sum of the distances to the points in $S(r)$. Finally, in the "furthest point outside the neighborhood" heuristic (FS), the point that has the largest sum of the distances to the points in $S(r)$ is selected.

1. $S(r) \leftarrow x_k$, where x_k is a random seed
2. $W \leftarrow V \backslash \{x_j: x_j \in N(r, x_k)\}$
3. While $|W| > 0$ do
   - 3.1. select a new point $x_k \in W$
   - 3.2. $W \leftarrow W \backslash \{x_j: x_j \in N(r, x_k)\}$
   - 3.3. $S(r) \leftarrow S(r) \cup \{x_k\}$

Fig. 4. Pseudo-code for the neighborhood algorithms. Different selection rules in step 3.1 give rise to different algorithms.

2.3. *Projection algorithm*

Wang and Kuo [5] describe a polynomial algorithm to solve the one-dimensional p-dispersion problem optimally. We can project the points of a Euclidean p-dispersion problem on a line, solve the problem on the line using the Wang–Kuo algorithm, and project the solution back to the Euclidean space. This approach is similar to the use of the spacefilling curve to solve combinatorial problems in the Euclidean space by first mapping the problem on a line, and then solving the problem on the line [12]. However, this heuristic is "arbitrarily bad". Consider a simple two-dispersion problem with 3 points. Suppose point 1 is located at $(-1, 5)$, point 2 is located at $(0, 0)$, and point 3 is located at $(1, 5)$. Clearly, the optimal solution consists of points 1 and 2, or points 2 and 3. If we project these three point on the x-axis, the projections will be at points $(-1, 0)$, $(0, 0)$ and $(1, 0)$ for point 1, 2 and 3, respectively. The optimal solution to the projected problem consists of the projections of points 1 and 3. However, this is the worst solution to the two-dimensional problem. Although a single application of the projection heuristic may give an arbitrarily bad solution, it may be possible to find a good solution to the Euclidean problem by using a number of projections. Figure 5 shows a pseudo-code for the projection algorithm, where step 2 is the solution of the one-dimensional p-dispersion problem [5].

1. Construct V' by projecting V on a line making an angle of α with the x-axis [projection of the point (x, y) is $x \cos \alpha + y \sin \alpha$]
2. Solve the one-dimensional problem on V'
3. Find the solution in V that corresponds to the solution in V'

Fig. 5. The pseudo-code for the projection algorithm.

2.4. *Interchange algorithms*

Among the three interchange heuristics, the first pairwise interchange heuristic (IF) and the best pairwise interchange heuristic (IM) are quite similar. They both start with a random solution S to the problem, and attempt to improve the objective function value by interchanging a point x in the solution with another point y not in the solution. The point x to be excluded from the solution is one of the two points x_1, x_2 in S that are the closest. Algorithm (IF) performs the interchange with the first point outside S that improves the objective function value, whereas algorithm (IM) considers all possible interchanges between x_1, x_2 and points outside S, and performs the interchange that improves the objective the most. Thus (IM) can be viewed as a greedy interchange algorithm. Clearly, these improvement heuristics can be used in conjunction with the other heuristics discussed so far. Figures 6 and 7 give the pseudo-codes for these two algorithms.

1. S a random solution
2. Perform the following loop until no change occurs in S
   - 2.1. Find x_1, x_2, with $x_1 \neq x_2$ and $d(x_1, x_2) = \min\{d(x_i, x_j): x_i, x_j \in S\}$
   - 2.2. For all points x_k in $V \backslash S$ do until a change occurs in S
     - 2.2.1 $S' \leftarrow \{S \backslash \{x_1\}\} \cup \{x_k\}$, $S'' \leftarrow \{S \backslash \{x_2\}\} \cup \{x_k\}$
     - 2.2.2. If $f(S') > f(S)$ and $f(S') > f(S'')$ then $S \leftarrow S'$
     - 2.2.3. If $f(S'') > f(S)$ and $f(S'') > f(S')$ then $S \leftarrow S''$

Fig. 6. The pseudo-code for the first pairwise interchange heuristic (IF).

1. S a random solution
2. Perform the following loop until no change occurs in S
   - 2.1. Find x_1, x_2, with $x_1 \neq x_2$ and $d(x_1, x_2) = \min\{d(x_i, x_j): x_i, x_j \in S\}$
   - 2.2. $SS \leftarrow S$
   - 2.3. For all points x_k in $V \backslash S$ do
     - 2.3.1. $S' \leftarrow \{S \backslash \{x_1\}\} \cup \{x_k\}$, $S'' \leftarrow \{S \backslash \{x_2\}\} \cup \{x_k\}$
     - 2.3.2. If $f(S') > f(SS)$ and $f(S') > f(S'')$ then $SS \leftarrow S'$
     - 2.3.3. If $f(S'') > f(SS)$ and $f(S'') > f(S')$ then $SS \leftarrow S''$
   - 2.4. $S \leftarrow SS$

Fig. 7. The pseudo-code for the best pairwise interchange heuristic (IM).

The interchange algorithms discussed above find two-optimal solutions, namely solutions that are optimal with respect to pairwise interchanges. They may, however, be stuck at a local maximum. One way to overcome this is by exchanging more than one point at a time. However, this is quite time consuming, since one would consider exchanging all pairs of points in S with all pairs of points not in S. Another way of getting out of a local maximum is by moving to an inferior solution. However, the interchange algorithms (IF) and (IM) are not equipped for this. In contrast, simulated annealing randomizes the improvement procedure somewhat by allowing occasional decreases in the objective function value. It can be shown that a simulated annealing algorithm converges to the global optimum [13]. However, the rate of convergence is exponential, which provides little hope that the optimum can be reached quickly. The empirical efficiency of a simulated annealing algorithm thus depends on the problem structure and on the choice of algorithm parameters. Our simulated annealing heuristic follows the generic code given in Ref. [14].

Suppose S is the current solution set and S' is a solution set obtained by performing a random pairwise interchange. If $f(S') > f(S)$, then we replace S by S'. On the other hand, if $f(S') \leqslant f(S)$, then we replace S by S' with a probability q, where q is a decreasing function of $[f(S) - f(S')]$. The motivation for moving to an inferior solution is that if we are stuck at a local maximum, the only way to obtain future improvements is by accepting a reduction in the objective function value at the current iteration. We are more likely to accept small reductions than large reductions in the objective, since the probability of accepting a reduction decreases with the size of the reduction. The probability also decreases with the number of iterations. One reason for this is to obtain convergence; another reason is that, as the global maximum is approached, reducing the objective becomes less attractive. Figure 8 summarizes our simulated annealing algorithm in pseudo-code format. We used the following parameters in our experiment:

$$\text{SIZEFACTOR} = 10*(n-p),\ \text{CUTOFF} = 10,\ \text{TEMPFACTOR} = 0.95,\ \text{MINPERCENT} = 2,\ \text{FREEZE\_LIM} = 5.$$

1. S is a random solution;
   FREEZE_LIM, SIZEFACTOR, CUTOFF, TEMPFACTOR, MINPERCENT are parameters
   freezecount $\leftarrow 0$, $T \leftarrow$ largest entry in the distance matrix
2. While freezecount < FREEZE_LIM do
   2.1. Changes = trials = 0;
   2.2. While trials < SIZEFACTOR and changes < CUTOFF do
      2.2.1. Trials $\leftarrow$ trials + 1
      2.2.2. S': a solution obtained from S by a random pairwise int.
      2.2.3. $e = f(S') - f(S)$, *rnd* a (0, 1) random number
      2.2.4. If $e > 0$ or *rnd* < exp(e/T) then $S \leftarrow S'$
   2.3. $T \leftarrow T$*TEMPFACTOR
   2.4. If $f(S)$ has increased then freezecount $\leftarrow 0$
   2.5. If changes/trials < MINPERCENT then freezecount $\leftarrow$ freezecount + 1

Fig. 8. The pseudo-code for the simulated annealing heuristic (SA).

3. A COMPARISON OF THE HEURISTICS

There are eight criteria for judging heuristics: simplicity, reasonable core storage requirements, speed, accuracy, robustness, ability to accept multiple starting points, ability to produce multiple solutions and good stopping criteria [15]. Before focusing on a comparison based on accuracy and speed, we briefly discuss some of the other criteria:

- The construction, deletion and interchange heuristics we consider are conceptually very simple. The neighborhood algorithms are somewhat more complicated because of the use of an external concept, the neighborhood, where the selection of the neighborhood radius is not trivial. Furthermore, they can be viewed as "incomplete", since they may not generate a solution to the p-dispersion problem, and may require an additional (addition or deletion) step. The projection algorithm relies on a one-dimensional p-dispersion algorithm, which is reasonably complicated. Simulated annealing is perhaps the most complicated algorithm, due to the parameters which must be fine-tuned by the user.

- Although some of the heuristics require only the distance matrix, others require a sorted distance array. Core storage requirements are of size $O(n^2)$ for all of these heuristics, which is quite reasonable.
- The three interchange heuristics and the three neighborhood algorithms can accept multiple starting points. A starting point for an interchange heuristic is a set of p points, whereas a starting point for a neighborhood heuristic is a single point (a seed). Since there are more sets of p points than the number of points, one may argue that the interchange algorithms are the most versatile from this perspective.
- All algorithms, except the greedy ones, can generate multiple solutions. To generate multiple solutions with the algorithms (FP), (IF), and (SA) one can change the indexing of the given points, since these algorithms operate on a first-point-satisfying-a-criterion basis.
- All of the agorithms, with the exception of (SA), have well-defined stopping criteria. For (SA), the stopping criteria is part of a fine-tuning process through parameter adjustments. The neighborhood algorithms usually require multiple applications due to the binary search over the neighborhood radius. One should definitely apply (SG) and (PR) several times, since a single application may result in a poor solution.

We now describe our computational experiments. We are interested in the quality of the solutions obtained, the time required to obtain these solutions, and the variation in the quality and the time. We tested the ten heuristics on a set of randomly generated Euclidean problems. We considered different values of n and p to observe the performance of the heuristics over different problem sizes. We used all combinations of $n=30$, 50, 100, 150, 200, 250 and 300, and $p=0.1n$, $0.3n$, $0.5n$, $0.7n$, $0.9n$. We solved 30 problems per set, resulting in a total of 1050 problems.

Algorithms (GC), (GD), (SA), (CS), (FS), (IF) and (IM) were applied once per problem, (SG) was applied $(n-p)$ times, (FP) was applied five times (with different indexing each time), (PR) was applied four times (projecting on the x-axis, y-axis, and on the lines $y=x$ and $y=-x$).

The code was written in $C++$ and run with BorlandC 2.0 on a 486 AT-compatible microcomputer. Our problems were generated randomly in R^2, with each coordinate uniformly distributed over (0, 100). We used the random number generator "rand()" in the standard library of C, initializing it with the seed k for the kth problem of a set of problems. After generating the points, the code computed the distance matrix using Euclidean distances, and generated a sorted distance array. We used an efficient heap sort routine, and our maximum sorting time was about 16 s for an array of 44,850 elements. We used the clock function "clock()" in C to find the number of clock ticks between two events. To convert this to seconds, we divided this number by CLK_TCK, a compiler dependent constant, which is 18.2 for BorlandC 2.0. The accuracy of the computational times is to 1/18.2 of a second.

The solution accuracy of each heuristic is evaluated relative to the best solution found for the problem. For $n=30$ and $n=50$, we found the optimal solutions using an implicit enumeration algorithm [2]. For the remaining problem sizes, the accuracy comparisons are based on the best solution found by the heuristics, since for $n \geqslant 100$ the implicit enumeration algorithm becomes too time-consuming. For each heuristic, we stored the percentage accuracy and the computational time for each problem in a set, and then printed the minimum, maximum, mean and standard deviation of these statistics.

In this paper, we concentrate on only a portion of our computational results due to space limitations. We give the statistics of the minimum (min%) and average (avg%) percentage of the solutions relative to the optimal solution, and the number (No.) of problems solved optimally for $n=30$ and $p=3$, 9, 15 in Table 1, and for $n=50$ and $p=5$, 15, 25 in Table 2. We turn to larger problems in Table 3, summarizing the minimum and average accuracies (as a percentage of the best solution found) for $n=100$ and $n=200$, and $p=0.1n$ and $p=0.3n$. We do not report the results for larger values of p in this paper for two reasons. Almost uniformly, all algorithms are more accurate for larger values of p. This is because of the multiplicity of optimal solutions for large values of p. For example, for $n=30$ and $p=27$, the least number of problems solved optimally for any algorithm (except the projection algorithm) is 26 out of 30. Furthermore, we believe that in most applications of the p-dispersion problem, p will be quite small in relation to n.

Table 1. Accuracy statistics for 30 problems with $n=30$ and $p=3, 9, 15$

	$p=3$			$p=9$			$p=15$		
Algorithm	mon (%)	avg (%)	No.	min (%)	avg (%)	No.	min (%)	avg (%)	No.
GC	71.9	88.0	0	76.4	89.6	3	76.8	91.8	2
GD	69.2	89.5	7	76.4	89.1	1	80.9	97.3	14
SG	80.0	94.8	13	79.8	90.8	2	80.3	96.1	12
FP	67.2	88.0	1	82.6	92.2	1	81.0	96.1	10
CS	52.4	72.3	0	68.7	88.7	3	76.2	94.2	7
FS	70.3	85.9	1	80.5	92.6	4	85.9	95.1	5
PR	58.0	79.2	1	39.2	53.4	0	34.9	53.7	0
IF	82.6	96.5	17	76.7	93.1	9	87.1	98.0	18
IM	77.0	95.8	16	76.5	93.0	7	80.9	96.8	16
SA	94.6	99.7	27	94.6	99.5	24	90.7	99.3	23

Table 2. Accuracy statistics for 30 problems with $n=50$ and $p=5, 15, 25$

	$p=5$			$p=15$			$p=25$		
Algorithm	min (%)	avg (%)	No.	min (%)	avg (%)	No.	min (%)	avg (%)	No.
GC	77.0	95.7	14	75.2	87.6	1	77.7	92.2	5
GD	61.0	82.0	1	77.5	89.5	0	79.9	94.9	7
SG	79.5	90.1	2	76.6	89.4	0	82.9	94.2	5
FP	78.8	88.4	0	80.7	90.6	1	84.8	96.1	8
CS	62.8	81.3	0	79.3	91.2	1	84.9	94.6	2
FS	75.8	91.1	4	80.6	92.5	2	83.1	95.7	7
IF	78.7	91.5	5	81.6	92.1	1	87.0	96.0	8
IM	78.7	91.2	5	77.2	91.7	5	79.9	94.9	7
SA	83.8	97.4	19	90.3	97.8	14	90.3	99.1	23

Table 3. Accuracy statistics for 30 problems with $n=100$ and 30 problems with $n=200$ with $p=0.1n$ and $0.3n$

	$n=100$				$n=200$			
	$p=10$		$p=30$		$p=20$		$p=60$	
Algorithm	min (%)	avg (%)	min (%)	avg (%)	min (%)	avg (%)	min (%)	avg (%)
GC	73.8	83.6	82.4	88.2	84.1	88.2	84.3	90.1
GD	63.2	81.0	80.7	90.9	73.2	80.7	85.9	90.5
SG	78.4	84.2	83.8	88.4	77.3	82.4	84.1	88.0
FP	81.3	87.4	83.7	91.4	82.7	88.8	88.4	92.3
CS	77.5	88.4	86.6	92.8	81.9	92.2	87.5	95.8
FS	86.2	93.4	88.4	96.3	90.3	97.0	90.1	97.5
IF	70.5	90.0	83.4	95.4	82.6	92.2	85.6	93.8
IM	77.1	88.6	83.4	95.1	83.5	93.1	86.3	93.8
SA	83.7	97.7	88.6	98.5	92.1	98.3	96.4	99.3

We left out from Tables 2 and 3 the statistics for the projection algorithm. This was due to its poor performance, which was the most striking outcome of the experiments. The solutions generated by this algorithm are clearly inferior to the solutions generated by any other algorithm. The four projections we use take up the same order of computational time as the other algorithms, but the solutions do not measure up. We note that the p-dispersion problem is very sensitive to a poor selection of points, since the objective function is equal to the smallest distance between all selected points. If one pair of close points are selected, it does not matter how good the rest of the solution is. When we project the problem on a line, a significant amount of distance information is lost. All distances are contracted, some more than others. Apparently, the algorithm is choosing points that seem to be separated on the line (relatively speaking), but are actually quite close in the plane.

Another result of our experiments is that (SA) is consistently the most accurate algorithm, and is also the most robust algorithm (standard deviations are less than 2% for all problem sizes reported in Table 1). For the 300 problems solved with $n=30$ and 50, the average accuracy of (SA) is 99.2%. Given that all three interchange algorithms were applied using a single random starting

solution, their solutions are surprisingly good. It is interesting to note that (IF) consumes less time than (IM), yet produces consistently better results.

One may expect (GC) to do well for problems with small p. Although this is the case in Table 2, Tables 1 and 3 paint a different picture. In fact, the average accuracy of this algorithm for $p=0.1n$ in larger problems ($n=200, 250, 300$) is never above 90%, and is always below the average accuracies of (CS), (FS), (IF) and (IM).

Algorithm (GD) seems well-suited for problems with relatively large p. It is, however, quite prone to producing a poor solution in problems with a relatively small p. Its minimum accuracy is the worst in many problem sets. The other deletion algorithm, (SG), seems unable to find "very good" solutions, although its average accuracies are reasonably good. It also requires relatively longer computational times, due to its large number of repetitions.

Among the neighborhood algorithms, (FS) is consistently superior to (CS) in terms of accuracy statistics. We note here that algorithm (FS) performs quite well in the sets we do not report here in detail. In fact, its average accuracy is above 96% in every one of these 25 problem sets. Note that in (CS) and (FS), the "closest" or the "furthest" point to the set S is defined in terms of the sum of the distances to the points in set S. This may not be the most natural definition of "closest" and "furthest". Usually, the distance between a point p and a set of points S is taken to be the minimum of the distances between point p and the points in S. Clearly, this interpretation of proximity would have an impact on the neighborhood algorithms. We experimented with these variants of (CS) and (FS), which we denote (CM) and (FM), respectively. The average accuracy of (CM) is better than the average accuracy of (CS) in only seven of the 35 sets in our experiment, and the average accuracy of (FM) is better than the average accuracy of (FS) in only five of the 35 sets in our experiment. In all these cases, the difference is quite insignificant (less than 1%). However, in some cases, (FS)'s average accuracy is better than (FM)'s by 10%. Thus, we conclude that, although somewhat counterintuitive, defining proximity in relation to all points in S (as was originally proposed by Steuer [4]) results in more effective heuristics than defining proximity in relation to the closest point in S.

Our experiment indicates that the neighborhood and the interchange algorithms are slightly more accurate than the construction algorithms. The accuracies of the neighborhood and interchange algorithms can be improved further by using multiple seeds, at a cost of increased computational times. We have very limited experience with multiple applications of these heuristics. As indicated in Table 1, for $(n, p)=(30, 3)$, (FS)'s average accuracy with one repetition is 85.9%. The average accuracy goes up to 90.3% with five repetitions, and to 93.9% with eight repetitions. The number of problems solved optimally goes from 1 with one repetition, to 6 with five repetitions, and to 11 with eight repetitions. Similarly, the average accuracies of (IF) and (IM) from Table 1 are 96.5 and 95.8%, respectively. If we run these algorithms with six different random starts, the average accuracies go up to 99.2 and 99.7%, respectively. As one would expect, the marginal increase in the average accuracy, as a result of an additional run, decreases with the number of runs. However, given the low computational requirements of the heuristics, multiple applications seem to be a good idea.

One can complement a construction or a neighborhood heuristic with an improvement phase by first finding the best solution with the construction or neighborhood heuristic, and then initializing an interchange heuristic with this best solution. We experimented with this strategy on our test problems, and found that almost all solutions were improved as a result of the improvement phase. Nearly all average optimality figures for the construction and neighborhood algorithms, complemented by (IF) and (IM), are above 95%, and many more optimal solutions are found. Due to space limitations, we do not report detailed accuracy statistics for these improved versions of the heuristics, but we do provide two examples. For example, for $n=30$ and $p=3$, the (GC) produces zero optimal solutions and an average accuracy of 88.0%. However the improved version of (GC) produces 24 optimal solutions, with an average accuracy of 98.6%. At the same time, the increase in the computational times, as a result of the addition of the improvement phase, is very marginal. To give another example, for $n=50$ and $p=5$, the minimum and average accuracy figures for (FS) are 75.8 and 91.1%, respectively. In contrast, for the improved version of (FS), the minimum and average accuracy figures are 84.5 and 95.9%, respectively, at an additional average time of about 1 s.

It is important to note here that starting an interchange algorithm with a good solution (namely

the best solution of another heuristic) does not always result in a better final solution than starting the interchange algorithm with a random solution. We do not suggest using the interchange algorithms just to complement the others. In some cases, they may be more effective as stand-alone algorithms. However, if one wishes to use anything but an interchange algorithm, it seems like a good idea to complement the algorithm with an interchange improvement phase.

The average computational times for all algorithms, except (SA), are all below 0.5 s for $n=30$ and all below 1.5 s for $n=50$. Algorithm (SA) takes two-to-three orders of magnitude longer than the other algorithms. As p is increased, (SA)'s times go down considerably, since fewer interchanges are performed for larger values of p. However, for large n and $p=0.1n$, the computational times become significant. For example, for problems with $(n, p)=(200, 20)$, (SA) consumes an average of about 5 min, while most other heuristics consume an average of 10 s. It may be possible to reduce (SA)'s computational times by changing its parameters. However, this may impact its accuracy adversely.

For our largest problems ($n=300$), the average computational times range from well under 10 s for (GD) to over 10 min for (CS) and (FS). For these large problems, (FP) produces almost as accurate results as (CS) and (FS) at a fraction of the effort. For a fixed n, as p is increased, interchange algorithms become computationally more attractive than neighborhood algorithms, since the neighborhood algorithms consume more time, while the interchange algorithms consume less time.

4. CONCLUSIONS

In this paper, we report our computational experience with 10 heuristic procedures for the p-dispersion problem. The only heuristic algorithm that generates poor solutions consistently is the projection heuristic. We have some very encouraging computational results with simulated annealing, which is consistently the most accurate heuristic in our experiment. However, given the effectiveness of the other heuristics, the effort necessary for fine-tuning the parameters of a simulated annealing heuristic may not be justified. Furthermore, this algorithm consumes significantly more computational time than the other algorithms.

Among the other heuristics, interchange and neighborhood heuristics seem to do better than construction heuristics. The greedy construction and deletion heuristics suffer from being shortsighted. They are also unable to accept multiple starting solutions and generate multiple solutions. The semi-greedy deletion heuristic, although able to generate reasonably good solutions very quickly, does not compete well against the interchange and neighborhood heuristics.

Among the neighborhood heuristics, the most effective one seems to be (FS), the furthest point outside the neighborhood heuristic. Among the two simple interchange heuristics, the first pairwise interchange heuristic (IF) appears to be the more effective and efficient one. The comparison between (FS) and (IF) is inconclusive. For the smaller problems ($n=30$), (IF) generates better solutions than (FS). For most of the larger problems, (FS) generates better solutions than (IF). However, (IF) is simpler to understand and to code, requires less memory, and is faster, with the differences in speed becoming more emphasized with increasing p.

A potential of the interchange and neighborhood algorithms we did not fully exploit in this experiment is their ability to generate multiple solutions through multiple starts. If one wishes to maximize effectiveness, one should apply the heuristics several times. Since the problem seems to get easier with increasing p, the number of trials should probably decrease with p. We did not detect a deterioration in the solutions as n was increased. Thus, we do not believe that it is necessary to increase the number of trials with n.

We suggest that a construction or neighborhood heuristic be complemented with an interchange heuristic. In our problem sets, this practice has always resulted in increased average accuracy. In the 150 problems we solved with $n=30$ (and different values of p), eight repetitions of the furthest point outside the neighborhood algorithm complemented with an improvement phase resulted in an average accuracy of 99.74%.

Considering all relevant criteria, it is not easy to declare one heuristic as being the overall "best" one for the discrete p-dispersion problem. However, we believe that this is a minor issue. Given how easy most of these heuristics are to code, and how reasonable the computational times are for most of them as well, we would suggest using several of them in parallel. In fact, for problems

with $n \leqslant 50$, we would recommend using all of them (save the projection algorithm) in parallel, complementing the construction and the neighborhood algorithms with improvement phases. This results in a composite heuristic (a "super" heuristic) with excellent accuracy. In the six problem sets summarized in Tables 1 and 2, the average optimality of this composite heuristic is 99.90%, and 170 of the 180 problems in these sets are solved optimally by this heuristic. This composite heuristic consumes between 2 and 80 s for the problems in these sets. If this is deemed too long, then we would suggest using all of the heuristics, except the most time-consuming one, namely (SA). This results in an average accuracy of 99.68%, solving 156 of the 180 problems optimally, and consuming between 1 and 12 s per problem. (We point out that this version of the composite heuristic is consistently superior to algorithm (SA) and uses less computational effort.)

Although the p-dispersion problem is known to be NP-hard, and although we cannot expect to design a heuristic for it guaranteeing a solution less than twice the optimum, it seems that we can find optimal, or near-optimal, solutions to the problem very frequently by using simple heuristics. Based on our experiment with the planar version of this problem with Euclidean distances, we believe that it is highly unlikely for the best solution found to be poor, if one performs repeated applications of a neighborhood heuristic, complemented with a pairwise interchange improvement heuristic.

Acknowledgements—This research has been supported in part by the Natural Sciences and Engineering Research Council of Canada (OGP 25481). This article resulted from a B.S. graduation project in the Department of Industrial Engineering at Boğaziçi University, Istanbul. We thank an anonymous referee whose comments resulted in improvements to the article.

REFERENCES

1. M. J. Kuby, Programming models for facility dispersion: the p-dispersion and maxisum dispersion problems. *Geog. Anal.* **19**, 315–329 (1987).
2. E. Erkut, The discrete p-dispersion problem. *Eur. J. Opl Res.* **40**, 48–60 (1990).
3. R. E. Steuer, *Multiple Criteria Optimization: Theory, Computation, and Application*, pp. 311–321. J. Wiley, New York (1986).
4. R. E. Steuer and F. W. Harris, Intra-set point generation and filtering in decision and criterion space. *Computers Ops Res.* **7**, 41–53 (1980).
5. D. W. Wang and Y. S. Kuo, A study of two geometric location problems. *Inf. Proc. Lett.* **28**, 281–286 (1988).
6. E. Erkut and S. Neuman, Comparison of four models for dispersing facilities. *INFOR* **29**, 68–86 (1991).
7. D. J. White, The maximal dispersion problem and the "first point outside the neighborhood" heuristic. *Computers Ops Res.* **18**, 43–50 (1991).
8. D. J. White, The maximal-dispersion problem. *IMA Jl Math. App. Bus. Ind.* **3**, 131–140 (1991).
9. S. S. Ravi, D. J. Rosenkrantz and G. K. Tayi, Facility dispersion problems: heuristics and special cases. *Lecture Notes Comp. Sci.* **519**, 355–366 (1991).
10. S. S. Ravi, D. J. Rosenkrantz and G. K. Tayi, Heuristic and special case algorithms for dispersion problems. *Ops Res.* **42**, 299–310 (1994).
11. R. K. Kincaid, Good solutions to discrete noxious location problems with metaheuristics. *Ann. Ops Res.* **40**, 265–281 (1992).
12. J. J. Bartholdi and L. K. Platzman, Heuristics based on spacefilling curves for combinatorial problems in Euclidean space. *Mgmt Sci.* **34**, 291–305 (1988).
13. G. L. Nemhauser and L. A. Wolsey, *Integer Combinatorial Optimization*, p. 407. J. Wiley, New York (1988).
14. D. S. Johnson, C. R. Aragon, L. A. McGeoch and C. Schevon, Optimization by simulated annealing: an experimental evaluation; Part II, graph coloring and number partitioning. *Ops Res.* **39**, 378–406 (1991).
15. S. H. Zanakis and J. R. Evans, Heuristic "optimization": why, when, and how to use it. *Interfaces* **11**, 84–91 (1981).