Dispersion on the Line

Toshihiro Akagi[1,a)] Shin-ich Nakano[1,b)]

Abstract: The facility location problem and many of its variants have been studied. A typical problem is to find a set of locations to place facilities with the designated cost minimized. By contrast, in this paper we consider the dispersion problem, which finds a set of locations with some objective function maximized. Given a set P of n possible locations, and a distance d for each pair of locations, and an integer p with $p \leq n$, we wish to find a subset $S \subset P$ with $|S| = p$ such that the objective function $f(S) = \min_{u,v \in S}\{d(u,v)\}$ is maximized. The intuition of the problem is as follows. Assume that we are planning to open several chain stores in a city. We wish to locate the stores mutually far away from each other to avoid self-competition. So we wish to find p locations such that the minimum distance between them is maximized. In this paper when P is a set of points on the line we first give a simple $O(n \log n)$ time algorithm, then a faster $O(n \log \log n)$ time algorithm, by the sorted matrix search method.

Keywords: facility location, dispersion

1. Introduction

The facility location problem and many of its variants have been studied[5], [6]. A typical problem is to find a set of locations to place facilities with the designated cost minimized. By contrast, in this paper we consider the dispersion problem, which finds a set of locations with some objective function maximized.

Given a set P of n possible locations, and a distance d for each pair of locations, (we assume d satisfy the triangle inequality,) and an integer p with $p \leq n$, we wish to find a subset $S \subset P$ with $|S| = p$ such that the objective function $f(S) = \min_{u,v \in S}\{d(u,v)\}$ is maximized. This is the Max-Min version of the dispersion problem[14]. For the Max-Sum version see [9], [12], for geometric version see[3] and for a variety of related problems see[4].

The intuition of the problem is as follows. Assume that we are planning to open several chain stores in a city. We wish to locate the stores mutually far away from each other to avoid self-competition. So we wish to find p locations so that the minimum distance between them is maximized. See more applications, including *result diversification*, in [10], [12], [13].

Paper [12], [14] shows if P is a set of points on the plane then the problem is NP-hard, and if P is a set of points on the line then the problem can be solved in $O(\max\{n \log n, pn\})$ time by dynamic programming approach. In this paper when P is a set of points on the line we first give a simple $O(n \log n)$ time algorithm, then a faster $O(n \log \log n)$ time algorithm, by the sorted matrix search method[7]. (See the good survey for the sorted matrix search method in [2], Section 3.3, or our explanation in [1]).

The remainder of the paper is organized as follows. Section 2 gives an algorithm to solve a decision version of the dispersion problem, which is used as a subroutine in the algorithms in Section 3 and 4. Section 3 contains our first algorithm for the dispersion problem. Section 4 gives our second algorithm for the dispersion problem. Finally Section 5 is a conclusion.

2. (k,p)-dispersion on the line

In this section we give a linear time algorithm to solve a decision version of the dispersion problem. This algorithm is used as a subroutine in the algorithms in Section 3 and 4.

Given possible locations $P = \{c_1, c_2, \cdots, c_n\}$ on the line (we assume they are distinct points and appear in those order from left to right, respectively) and two numbers p and k, then we with to know if there exists a subset $S \subset P$ with $|S| = p$ such that $f(S)$, which is defined as $\min_{u,v \in S}\{d(u,v)\}$, is greater than or equal to k.

The algorithm shown below is a greedy algorithm to solve the problem.

Algorithm 1 Test-dispersion(k,p)

```
 1: set count = 0
 2: set l = 1
 3: set r = 1
 4: while r < n do
 5:     while r < n and d(c_l, c_r) < k do
 6:         r = r + 1
 7:     end while
 8:     if d(c_l, c_r) ≥ k then
 9:         count = count + 1
10:         l=r
11:     end if
12: end while
13: if count ≥ p then
14:     Output YES
15: else
16:     Output NO
17: end if
```

[1] Departmento of Cumputer Science, Gunma University, Kityu, 376-8515, Japan
[a)] akagi@nakano-lab.cs.gunma-u.ac.jp
[b)] nakano@cu.gunma-u.ac.jp

Lemma 2.1 The algorithm is correct.

Proof. Assume for the contradiction that the algorithm output NO but it has a subset $S \subset P$ with $|S| = p$ such that $f(S)$ is at least k. Let i be the minimum i such that the i-th location chosen by our algorithm is greater than the i-th location from the left in S. However it contradicts to the greedy choice of our algorithm. □

Lemma 2.2 The running time of the algorithm above is $O(|P|)$.

Proof. Since it scans P once. □

3. p-dispersion on the line

One can design an $O(n \log n)$ time algorithm to solve the dispersion problem when all P are on the line, based on the sorted matrix search method[2], [7].

Our strategy is as follows. First we can observe that the maximum value $k^* = f(S)$ of a solutions of a dispersion problem is the distance between some $u \in P$ and $v \in P$. Since the number of such distances is at most n^2, sorting them needs $O(n^2 \log n^2) = O(n^2 \log n)$ time. Then, by binary search, find the largest k such that the (k, p)-dispersion problem has a solution, using the linear-time decision algorithm in the preceding section, $\log n^2 = 2 \log n$ times. This part needs $O(n \log n)$ time. Thus the total running time is $O(n^2 \log n)$.

However by using the sorted matrix searching method[7] (See the good survey in [2], Section 3.3) we can improve the running time to $O(n \log n)$, then improve farther to $O(n \log \log n)$ time in Section 4. Similar technique is also used for a fitting problem[8], [11], and a r-gathering problem[1]. Now we explain the detail.

First let M be the matrix in which each element is $m_{i,j} = x(c_j) - x(c_i)$, where $x(v)$ is the coordinate of point v. Then $m_{i,j} \leq m_{i,j+1}$ and $m_{i,j} \geq m_{i+1,j}$ always holds, so the elements in the rows and columns are sorted, respectively. The minimum cost k^* of a solution of the dispersion problem is some element in the matrix. We are going to find the smallest k in M for which the (k, p)-dispersion problem has a solution, as follows.

By appending a suitable number of large enough elements to M as the elements in the topmost rows and the rightmost columns we can assume n is a power of 2. Note that the elements in the rows and columns are still sorted, respectively. Let M be the resulting matrix. Our algorithm consists of stages $s = 1, 2, \cdots, \log n$, and maintains a set L_s of (non-overlapping) submatrices of M possibly containing k^*. Hypothetically first we set L_0={M}. Assume we are now starting stage s.

For each submatrix M in L_{s-1} we divide M into the four submatrices with $n/2^s$ rows and $n/2^s$ columns and put them into L_s. We never copy these submatrices. We just update the index of the corner elements of each submatrix.

Let k_{min} be the median of the lower left corner elements of the submatrices in L_s. Then for the $k = k_{min}$ we solve the (k, p)-dispersion problem, using the algorithm in Section 2. We have the following two cases.

If the (k, p)-dispersion problem has a solution then we remove from L_s each submatrix with the lower left corner element (the smallest element) greater than k_{min}. Since $k_{min} \geq k^*$ holds each removed submatrix has no chance to contain k^*. Also if L_s has several submatrices with the lower left corner element equal to k_{min} then we remove them except one from L_s. Thus we can remove at least $|L_s|/2$ submatrices from L_s.

Otherwise if the (k, p)-dispersion problem has no solution then we remove from L_s each submatrix with the upper right corner element (the largest element) smaller than k_{min}. Since $k_{min} < k^*$ holds each removed submatrix has no chance to contain k^*. Now we can observe that, for each "chain" of submatrices, which is the sequence of submatrices in L_s with lower left to upper right diagonals on the same line, the number of submatrices (1) having the lower left corner element smaller than k_{min} (2) but remaining in L_i is at most one (since the elements on "the common diagonal line" are sorted). Thus, if $|L_s|/2 > D_s$, where $D_s = 2^{s+1}$ is the number of chains plus one, then we can remove at least $|L_s|/2 - D_s + 1$ submatrices from L_s.

Similarly let k_{max} be the median of the upper right corner elements of the submatrices in L_s, and for the $k = k_{max}$ we solve the (k, p)-dispersion problem and similarly remove some submatrices from L_s. This ends stage s.

Now after stage $\log n$ each matrix in $L_{\log n}$ has just one element, then we can find the k^* using a binary search with the linear-time decision algorithm in Section 2.

We can prove that at the end of stage s the number of submatrices in L_s is at most $2D_s$, as follows.

First L_0 has 1 submatrix, which is less than $2D_0 = 4$. By induction assume that L_{s-1} has $2D_{s-1} = 2 \cdot 2^s$ submatrices.

At stage s we first partite each submatrix in L_{s-1} into four submatrices then put them into L_s. Now the number of submatrices in L_s is at most $4 \cdot 2D_{s-1} = 4D_s$. We have four cases.

If the (k, p)-dispersion problem has a solution for $k = k_{min}$ then we can remove at least a half of the submatrices from L_s, and so the number of the remaining submatrices in L_s is at most $2D_s$, as desired.

If the (k, p)-dispersion problem has no solution for $k = k_{max}$ then we can remove at least a half of the submatrices from L_s, and so the number of the remaining submatrices in L_s is at most $2D_s$, as desired.

Otherwise if $|L_s|/2 \leq D_s$ then the number of the submatrices in L_s (even before the removal) is at most $2D_s$, as desired.

Otherwise (1) after the check for $k = k_{min}$ we can remove at least $|L_s|/2 - D_s$ submatrices (consisting of too small elements) from L_s, and (2) after the check for $k = k_{max}$ we can remove at least $|L_s|/2 - D_s$ submatrices (consisting of too large elements) from L_s, so the number of the remaining submatrices in L_s is at most $|L_s| - 2(|L_s|/2 - D_s) = 2D_s$, as desired.

Thus at the end of stage s the number of submatrices in L_s is always at most $2D_s$, and at the end of stage $\log n$ the number of submatrices is at most $2D_{\log n} = 4n$.

Now we consider the running time up to here. We implicitly treat each submatrix as the index of the upper right element in M and the number of lows (= the number of columns). Except for the calls of the linear-time decision algorithm for the (k, p) dispersion problem, we need $O(|L_{s-1}|) = O(D_{s-1})$ time for each stage $s = 1, 2, \cdots, \log n$, and $D_0 + D_1 + \cdots + D_{\log n - 1} = 2 + 4 + \cdots + 2^{\log n} < 2 \cdot 2^{\log n} = 2n$ holds, so this part needs $O(n)$ time in total. (Here

we use the linear time algorithm to find the median.)

Since each stage calls the linear-time decision algorithm twice and the number of stage is $\log n$ this part needs $O(n \log n)$ time in total.

After stage $s = \log n$ each matrix has just one element. Then we can find the k^* among the $|L_{\log n}| \leq 2D_{\log n} = 4n$ elements by (1) sorting them, then (2) performing binary search with the linear-time decision algorithm at most $\log 4n$ times. This part needs $O(n \log n)$ time.

Thus the total running time is $O(n \log n)$.

Theorem 3.1 One can solve the dispersion problem in $O(n \log n)$ time when all P are on the real line.

4. Faster Algorithm

In this section we give a faster algorithm to solve the dispersion problem when all P are on the real line, using the $O(n \log \log n)$ time faster matrix search method[7].

We always update (narrow) the possible range $[k_{low}, k_{high}]$, in which the best value k^* of the objective function exists.

We first build two tables which is used in our faster algorithm to solve the decision version of the dispersion problem. First we construct a set of submatrices N of M, as follows. Partite P into subsets, called subpaths, each of which consists of $c \log n$ consecutive locations, where c is a constant. Let P_t be the t-th subpath, and N_t the matrix in which each element $n_t(i, j)$ is the distance between i-th location and j-th location in P_t. Let N be the set of $N_1, N_2, \cdots, N_{n/\log n}$.

Perform matrix search $s = \log c + \log \log n$ rounds for N. Now the size of each submatrix is 1 by 1, (since $c \log n/2^s = 1$ holds for $s = \log c + \log \log n$) and the number of remaining submatrices is at most $2D_s n/c \log n = 4 \cdot 2^s n/c \log n = 4n$. Thus the number of remaining distances in current N is at most $4n$.

Then perform ordinary binary search (compute the median k then check if it has a solution with cost k, then remove a half of distances) $s = 2 + 2 \log \log n$ rounds for the remaining at most $4n$ distances. Now the number of remaining distances is at most $4n/2^s = n/\log^2 n$. The running time upto here is $O(n \log \log n)$

We say subpath P_t is *active* if N still contain a distance in N_t, and *non-active* otherwise. Now the number of active subpaths is at most $n/(\log n)^2$. For each non-active subpath P_t we construct the following two tables $ncount(i, t)$ and $remainder(i, t)$.

Let P_t^i be the subpath of P_t starting at the i-th location of P_t and ending at the last location of P_t. Then $ncount(i, t)$ stores the maximum number of locations such that (1) the distance between them is at least k, where k is some number among $[k_{low}, k_{high}]$, and (2) includes the i-th location. Note that since all distances corresponding to a non-active subpath are out of $[k_{low}, k_{high}]$, so for any k in $[k_{low}, k_{high}]$ $ncount(i, t)$ is identical. Then $remainder(i, t)$ is the index of the rightmost location in the selected locations above. One can compute those tables for a non-active subpath in $O(\log n)$ time by dynamic programming from right to left. (We assume the line is horizontal and $c_1, c_2, \cdots$ appear left to right.) Thus we need $O(n)$ time for all subpaths.

Now we are ready for the faster test, which needs only $O(n/\log n)$ time for test once. The test scans each $P_1, P_2, \cdots$ in this order. We estimate the running time for scanning subpaths in two cases.

For an active subpath we first find the first location to choose in $O(\log n)$ time by greedy scan, then scan the rest of the subpath in $O(\log n)$ time to compute locations to be chosen. Since the number of active subpaths is at most $n/\log^2 n$ the total running time of these parts is $O(n/\log n)$.

For a non-active subpath we compute the first location to be chosen in $O(\log \log n)$ time by binary search, then by just looking up the table we compute (1) the number of locations to be chosen in the subpath and (2) the last location to be chosen. Thus the running time for scan a non-active subpath is $O(\log \log n)$ time, and for all non-active subpaths we need $O(n \log \log n/\log n)$ time.

Thus in total we need $O(n \log \log n/\log n)$ time for scan once. See our algorithm below.

Algorithm 2
PREPROCESSING-for-Faster-Test-dispersion(k,p)

```
/* Let n be the number of rows (and also columns) in P */
Partite P into n/log n subpaths each of which consisting of consecutive
  log n locations in P.
Let N_t be the distance matrix of the t-th subpath above,
  and N the set of them.
Set k_low = 0 and k_high = ∞
Perform Matrix Search log c + log log n rounds for N
  with updating k_low and k_high
/* Now the number of the remaining distances is at most 4n */
Perform binary Search 2 + 2 log log n rounds for the remaining distances.
/* Now the number of the remaining distances is at most n/log^2 n */
/* A subpath is active if it has one or more remaining distances,
  and non-active otherwise */
for each t = 1, 2, ..., n/log n do
    if t-th subpath is non-active then
        for each i = log n, log n − 1, ..., 1 do
            Compute ncount(t,i) and remainder(t,i)
        end for
    end if
end for
```

Algorithm 3 Faster-Test-dispersion(k,p)

```
for each t = 1, 2, ..., n/log n do
    if t-th subpath is active then
        find the first location to be chosen by scan
        then scan the rest of the subpath to compute locations to be chosen
    else
        /* Now t-th subpath is non-active */
        find the first location to be chosen by binary search
        compute the number of locations to be chosen by checking
        ncount(t, i)
        compute the last location to be chosen by checking remainder(t, i)
    end if
end for
if the number of chosen locations ≥ p then
    Output YES
else
    Output NO
end if
```

The main algorithm consists of $\log n$ rounds, and each round consists of the following three steps, (1) MEDIAN COMPUTATION (2) TEST and (3) UPDATE. First MEDIAN COMPUTATION needs $O(1 + 2 + 4 + ... + n) = O(n)$ time in total for the $\log n$ rounds. TEST once needs only $O(n \log \log n / \log n)$ time, and $O(n \log \log n)$ time in total for the $\log n$ rounds. UPDATE needs $O(1 + 2 + 4 + ... + n) = O(n)$ time in total for the $\log n$ rounds.

Thus in total the main part of the algorithm runs in $O(n \log \log n)$ time. We have the following theorem.

Theorem 4.1 One can solve the dispersion problem in $O(n \log \log n)$ time when all P are on the real line.

5. Conclusion

In this paper we have presented two algorithms to solve the dispersion problem when all P are on the real line. The running time of the second algorithm is $O(n \log \log n)$, which is faster than known algorithms.

Can we design a linear time algorithm for the dispersion problem when all P are on the real line?

References

[1] T. Akagi and S. Nakano, On r-Gatherings on the Line, Proc. of FAW 2015, LNCS 9130, pp. 25-32 (2015).
[2] P. Agarwal and M. Sharir, Efficient Algorithms for Geometric Optimization, Computing Surveys, 30, pp.412-458 (1998).
[3] C. Baur and S.P. Feketee, Approximation of Geometric Dispersion Problems, Pro. of APPROX '98, Pages 63-75 (1998).
[4] B. Chandra and M. M. Halldorsson, Approximation Algorithms for Dispersion Problems, J. of Algorithms, 38, pp.438-465 (2001).
[5] Z. Drezner, Facility Location: A Survey of Applications and Methods, Springer (1995).
[6] Z. Drezner and H.W. Hamacher, Facility Location: Applications and Theory, Springer (2004).
[7] G. Frederickson, Optimal Algorithms for Tree Partitioning, Proc. of SODA '91 Pages 168-177 (1991).
[8] H. Fournier, and A. Vigneron, Fitting a Step Function to a Point Set, Proc of ESA 2008, Lecture Notes in Computer Science, 5193, pp.442-453 (2008).
[9] R. Hassin, S. Rubinstein and A. Tamir, Approximation Algorithms for Maximum Dispersion, Operation Research Letters, 21, pp.133-137 (1997).
[10] T. L. Lei and R. L. Church, On the unified dispersion problem: Efficient formulations and exact algorithms, European Journal of Operational Research, 241, pp.622-630 (2015).
[11] J. Y. Liu, A Randomized Algorithm for Weighted Approximation of Points by a Step Function, Proc. of COCOA 2010, Lecture Notes in Computer Science, 6508, pp.300-308 (2010).
[12] S. S. Ravi, D.J. Rosenkrantz and G. K. Tayi, Heuristic and Special Case Algorithms for Dispersion Problems, Operations Research, 42, pp.299-310 (1994).
[13] M. Sydow, Approximation Guarantees for Max Sum and Max Min Facility Dispersion with Parameterised Triangle Inequality and Applications in Result Diversification, Mathematica Applicanda, 42, pp.241-257 (2014).
[14] D. W. Wang and Yue-Sun Kuo, A study on Two Geometric Location Problems, Information Processing Letters, 28, pp.281-286 (1988).