Efficient Algorithms for the 2-Center Problems

Tadao Takaoka

Department of Computer Science
University of Canterbury
Christchurch, New Zealand

Abstract. This paper achieves $O(n^3 \log\log n / \log n)$ time for the 2-center problems on a directed graph with non-negative edge costs under the conventional RAM model where only arithmetic operations, branching operations, and random accessibility with $O(\log n)$ bits are allowed. Here n is the number of vertices. This is a slight improvement on the best known complexity of those problems, which is $O(n^3)$. We further show that when the graph is with unit edge costs, one of the 2-center problems can be solved in $O(n^{2.575})$ time.

1 Introduction

The k-center problem is important in network analysis in various areas, such as facility location in operations research, location of file servers in local area networks, analyzing influential persons in social networks, etc. By identifying k-centers, we can minimize the distance or time from those centers to members of the given network under various criteria.

In this paper we consider the 2-center problems for a directed graph, which is regarded as the special case of the k-center problems with $k = 2$. Given a directed graph with non-negative real numbers as edge costs, the k-center problems compute the set of k vertices, called centers, which serve the whole graph in an optimal way in two definitions. One is to minimize the longest distance from the set of centers to all vertices. Here the distance from the set to a vertex v is the shortest distance from the k vertices to v. We call this problem the absolute k-center problem. The other definition is to minimize the sum of the shortest distances to all vertices from the k vertices in the set. This latter definition by the sum, if divided by n, is regarded as the average distance from the set to all vertices, and thus called the average k-center problem. These two problems are defined in Kariv and Hakimi [11], [12] and Gary and Johnson [8], and shown to be NP-complete. When $k = 2$, straightforward algorithms of $O(n^3)$ time are known for the two 2-center problems. In [11] and [12] $O(n^2 \log n)$ time and $O(n^2)$ time algorithm for the absolute and average k-center problems are shown for trees. Frederickson [6] gives a linear time algorithm for the k-center problems for a tree with unit edge costs. Apart from trees, if we restrict the graph to a cactus, where cycles share at most one vertex, $O(n \log^2 n)$ time algorithm is known for the absolute k-center problem [3].

We show in this paper those problems can be solved in $O(n^3 \log\log n/\log n)$ time for any directed graph after the all pairs shortest path (APSP) problem is solved in the same amount of time. We use the algorithm by the author [16], which is based on a fast algorithm for distance matrix multiplication (DMM). It is shown in page 204 of [1] that the time complexity of (n, n)-distance matrix multiplication (DMM) is asymptotically equal to that of the APSP problem for a graph with n vertices. Thus [16] mainly deals with DMM. The algorithm for DMM in [16] is based on the divide-and-conquer and table look-up approach.

Fredman [7] was the first to break the cubic bound $O(n^3)$ with $O(n^3(\log\log n/\log n)^{1/3})$ for the APSP problem. This complexity was improved to $O(n^3(\log\log n/\log n)^{1/2})$ by Takaoka [13] with RAM, and to $O(n^3/(\log n)^{1/2})$ by Dobosievicz [5] with extended logical operations. Since then, there have been some more progresses such as $O(n^3(\log\log n)/\log n)^{5/7})$ [9] and $O(n^3(\log\log n)^2/\log n)$ [15], and $O(n^3 \log\log n/\log n)$ [16]. Recently, algorithms with complexity $O(n^3\sqrt{\log\log n}/\log n)$ [18], $O(n^3(\log\log n/\log n)^{5/4})$ [10], $O(n^3(\log\log n)^3/\log^2 n))$ [4], etc., appeared.

Our algorithms for the 2-center problems are also based on matrix multiplication algorithms under different definitions, but the idea of using divide-and-conquer and table look-up is the same. The computational model in this paper is the conventional RAM, where only arithmetic operations, branching operations, and random accessibility with $O(\log n)$ bits are allowed all in $O(1)$ time. The basic idea in [7] and the subsequent improvements is that we speed up the computation by processing $O(\log n)$ bits in $O(1)$ time by either random accessibility with $O(\log n)$ bits or bit-wise logical operations on $\log n$ bits. We do not use bit-wise logical operations in this paper.

To multiply the small matrices resulting from dividing the original matrices, we sort distance data, merge sorted lists and use the ranks of those data in the merged lists. As the ranks are small integers, the multiplication can be done efficiently by looking at some pre-computed tables. We call this task of sorting "presort".

When edge costs are small integers, such as unit edge costs, we can solve the absolute 2-center problem in time more sub-cubic, that is, $O(n^{2.575})$. More precisely, $O(n^{2.575})$ is the time for solving the APSP problem with unit edge costs. Once the APSP problem is solved, the rest can be solved in $O(n^{2.376}\log n)$, where $O(n^{2.376})$ is the time for ordinary matrix multiplication. Thus we can say the APSP is the bottle neck for the absolute 2-center problem in this case.

The rest of the paper is as follows: Section 2 defines the two versions of k-center problems, and gives straightforward algorithms for them. Also the two versions of the 2-center problems are defined, and a formalization for solutions through matrix multiplication is given. In Section 3, we introduce the basic encoding scheme to deal with several small integers together in $O(1)$ time, which contributes to the speed-up of our algorithms. In Section 4, we review the divide-and-conquer algorithm for distance matrix multiplication given in [16] for two reasons. One is that the algorithm is used as the first stage of our 2-center algorithms. The other is that the technique used in [16], divide-and-conquer and

table look-up for small matrices, is extended to our problems in this paper. In Section 5, new definitions of matrix multiplication are given, which are used for our 2-center problems. In Section 6, we show how to construct tables used in our algorithms, and show that the times for constructing those tables are within the claimed time complexity. Section 7 summarises the whole algorithms based on the parts described in the earlier sections. Section 8 is devoted to an efficient algorithm for the absolute 2-center problem with edge costs of small integers. Section 9 discusses a possible application of the 2-center algorithms for k-center problems, and concludes the paper.

2 k-Center problems

Let $G = (V, E)$ be a directed graph with edge costs of non-negative real numbers. Vertices are given by integers between 1 and n, and edges are by pairs of vertices. Let $d(i, j)$ be the edge cost from vertex i to vertex j, and $d^*(i, j)$ be the edge cost of the shortest path from i to j. The shortest path from i to j is the path with the minimum sum of costs of edges over all possible paths. Let the matrices D and D^* be the matrices whose (i, j) elements are $d(i, j)$ and $d^*(i, j)$ respectively. The problem of computing D^* are known to be the all-pairs shortest path (APSP) problem and well studied. We use the algorithm in [16].

We define two kinds of k-center problems in this paper. Let C be a subset of V and $k = |C|$, the size of C. Let distance from C to vertex v, $dis(C, v)$, be defined by

$$dis(C, v) = \min\{d^*(u, v) | u \in C\}$$

If C is the set of fire stations, v is a house in a town, and edges are roads, $dis(C, v)$ is the distance from the nearest station to the house, although we deal with the more general model of directed graph. The cost of C is measured by the following two measures of $abs(C)$ and $ave(C)$.

$$abs(C) = \max\{dis(C, v) | v \in V\}$$

$$ave(C) = \Sigma_{v \in V} dis(C, v)$$

Finally the optimal k-center with absolute measure, C_{abs}, and that with average measure, C_{ave}, are defined by C that gives c_{abs} and c_{ave} in the following.

$$c_{abs} = \min\{abs(C) | C \subset V \& |C| = k\}$$

$$c_{ave} = \min\{ave(C) | C \subset V \& |C| = k\}$$

Intuitively speaking, C_{abs} is to minimize the longest distance from any fire station to houses while the number of stations is fixed to k. C_{ave} is to minimize

the average distance under the same condition. The actual average is c_{ave}/n. In [12], C_{ave} is known as p-medians.

Both problems are known to be NP-complete, and thus there will unlikely be any polynomial time algorithm. The following is a straight-forward algorithm of exponential time based on a fast APSP algorithm. In the following consecutive statements in the same indentation level are regarded as being in the scope of the preceding "for".

ALGORITHM 1 *k-Center with absolute measure*
1. Solve the APSP problem by any fast algorithm
2. $opt\_value = \infty$
3. for $C \subset V$ *such that* $|C| = k$ *do*
4. $abs = -\infty$
5. for $v \in V$ *do*
6. $dis = \infty$
7. for $u \in C$ *do* $dis = min\{dis, d^*(u, v)\}$
8. $abs = max\{abs,\ dis\}$
9. $opt\_value = min\{opt\_value,\ abs\}$
10. $c_{abs} = opt\_value$

The algorithm for the average measure can be derived with slight modification in the following.

ALGORITHM 2 *k-Center with average measure*
1. Solve the APSP problem by any fast algorithm
2. $opt\_value = \infty$
3. for $C \subset V$ *such that* $|C| = k$ *do*
4. $ave = 0$
5. for $v \in V$ *do*
6. $dis = \infty$
7. for $u \in C$ *do* $dis = min\{dis, d^*(u, v)\}$
8. $ave = ave + dis$
9. $opt\_value = min\{opt\_value,\ ave\}$
10. $c_{ave} = opt\_value$

In both algorithms, line 3, 5, and 7 are iterated $O(n^k)$, n, and k times respectively, resulting in total time of $O(M(n) + n^k kn)$, where $M(n)$ is the time for distance matrix multiplication. There are several algorithms with $M(n)$ less than $O(n^3)$, that is, sub-cubic. When $k = 2$, the time becomes $O(n^3)$. Our purpose is to improve the second term of complexity to be sub-cubic, so that the total time becomes sub-cubic.

We reformulate the problems for $k = 2$. Let us start from C_{abs}. Let the set of centers be $C = \{i, j\}$ and D' be the transposed matrix of D^*, that is, $d'(i, j) = d^*(j, i)$. We compute $abs(i, j)$ for all i and j by

$$abs(i, j) = \max_{k=1,n}\{\min\{d^*(i, k), d'(k, j)\}\}$$

Then c_{abs} can be computed in $O(n^2)$ time by

$$c_{abs} = min_{i=1,n;j=1,n}\{abs(i,j)\}$$

Let us define the (max, min)-product of matrices A and B by $P = AB$, where $P = \{p(i,j)\}$ is given by

$$p(i,j) = max_{k=1,n}\{min\{a(i,k), b(k,j)\}\}.$$

The set of $\{abs(i,j)\}$ for all i and j is given by D^*D' under (max, min)-matrix multiplication, abbreviated as (max, min)-multiplication. Note that if A and B are 0-1 matrices, this becomes Boolean matrix multiplication.

Similarly the 2-center with average measure can be reformulated. Let

$$ave(i,j) = \Sigma_{k=1,n}\{\min\{d^*(i,k), d'(k,j)\}\}$$

Then c_{ave} can be computed in $O(n^2)$ time by

$$c_{ave} = \min_{i=1,n;j=1,n}\{ave(i,j)\}$$

Let us define the (Σ, min)-product of matrices A and B by $Q = AB$, where $Q = \{q(i,j)\}$ is given by

$$q(i,j) = \Sigma_{k=1,n}\{min\{a(i,k), b(k,j)\}\}.$$

Then the set $\{ave(i,j)\}$ for all i and j is given by D^*D' under (Σ, min)-matrix multiplication, abbreviated as (Σ, min)-multiplication.

Thus our problem reduces to how fast we can compute the (max, min)-product and (Σ, min)-product for the given two matrices with non-negative real elements.

3 How to encode a short list of small integers

A short list of small integers bounded by μ, $\mathbf{x} = (x_1, x_2, ..., x_m)$, is encoded into a single integer $h(\mathbf{x})$ with $0 \leq x_i \leq \mu - 1$ for all i by

$$h(\mathbf{x}) = (x_1 - 1)\mu^{m-1} + ... + (x_{m-1} - 1)\mu + x_m - 1$$

Note that this function h is one-to-one, and those variable involved have values small enough that $h(\mathbf{x})$ is contained in a single word. The encoding can be done in $O(m)$ time by the Horner algorithm and decoding can be done in $O(m)$ time by successive division by μ. We use this encoding scheme with various variables in later sections. The maximum value of $h(\mathbf{x})$ is bounded by μ^m. Since $\mu^m = c^{m \log \mu}$ for some constant $c > 0$, we can choose m and μ such that

$m = O(\log n/\log\log n)$ and $\mu = O(\log n)$ to satisfy $h(\mathbf{x}) = O(n)$. We use the same name c for various constants hereafter.

Let us call h the packing function, since the encoding is like packing small integers into a single word.

4 Brief review of distance matrix multiplication

The normal product of two matrices A and B, $C = AB$, is defined by

$$c_{ij} = \Sigma_{k=1}^{n} a_{ik} \cdot b_{kj}, (i, j = 1, ..., n) \quad (1)$$

Now we use the divide-and-conquer approach, that is, divide A, B, and C into (m, m)-submatrices for $N = n/m$ as follows:

$$\begin{pmatrix} A_{1,1} & ... & A_{1,N} \\ ... & & \\ A_{N,1} & ... & A_{N,N} \end{pmatrix} \begin{pmatrix} B_{1,1} & ... & B_{1,N} \\ ... & & \\ B_{N,1} & ... & B_{N,N} \end{pmatrix} = \begin{pmatrix} C_{1,1} & ... & C_{1,N} \\ ... & & \\ C_{N,1} & ... & C_{N,N} \end{pmatrix}$$

Matrix C can be computed by

$$C_{ij} = \Sigma_{k=1}^{N} \{A_{ik} B_{kj}\} (i, j = 1, ...N) \quad (2)$$

where the product of submatrices is defined similarly to (1). This divide-and-conquer approach is made possible thanks to the associative property of the Σ operation.

The distance matrix multiplication is to compute the following distance product $C = AB$ for two (n, n)-matrices $A = [a_{ij}]$ and $B = [b_{ij}]$ whose elements are real numbers. In the following we replace the addition and multiplication by various operations in such a way that we can still take the divide-and-conquer approach. The first is min for Σ and $+$ for "$\cdot$" given as follows:

$$c_{ij} = \min_{k=1}^{n} \{a_{ik} + b_{kj}\}, (i, j = 1, ..., n) \quad (3)$$

$C_{ij} = \min_{k=1}^{N} \{A_{ik} B_{kj}\} (i, j = 1, ...N)$ (4) (Each multiplication is similar to (3))

The "min" operation is defined on submatrices by taking the "min" operation componentwise. This product is called distance product or $(min, +)$-product. We have N^3 multiplications of distance matrices in (4). Let us assume that each multiplication of (m, m)-submatrices can be done in $T(m)$ computing time, assuming pre-computed tables are available. The time for constructing the tables is reasonable when m is small. The time for min operations in (4) is $O(n^3/m)$ in total. Thus the total time excluding table construction is given by $O(n^3/m + (n/m)^3 T(m))$. By choosing an appropriate value for m, we can show the time for $(min, +)$-multiplication is $O(n^3 \log\log n/\log n)$. More details of the algorithm for $(min, +)$-multiplication is given in Appendix.

5 (max, min)-multiplication and (Σ, min)-multiplication

By replacing the Σ and $\cdot$ operations pair by (max, min) and (Σ, min), we define the following two more matrix products. We call those products (max, min)-product and (Σ, min)-product. The two products, the main theme of this paper, are defined by

$$c_{ij} = \max_{k=1}^{n}\{\min\{a_{ik}, b_{kj}\}\}, (i, j = 1, ..., n) \quad (5) \quad ((max, min)\text{-product})$$

$$C_{ij} = \max_{k=1}^{N}\{A_{ik}B_{kj}\}(i, j = 1, ...N) \quad (6) \quad \text{(Each multiplication is similar to (5))}$$

$$c_{ij} = \Sigma_{k=1}^{n}\{\min\{a_{ik}, b_{kj}\}\}, (i, j = 1, ..., n) \quad (7) \quad ((\Sigma, min)\text{-product})$$

$$C_{ij} = \Sigma_{k=1}^{N}\{A_{ik}B_{kj}\}(i, j = 1, ...N) \quad (8) \quad \text{(Each multiplication is similar to (7))}$$

Let us rename A_{ik} and B_{kj} in (6) by A and B. Let $M = \{1, ..., m\}$. Let $S(i, j)$ and $T(i, j)$ be defined by

$$S(i, j) = \{k|a_{ik} \leq b_{kj}\},\ T(i, j) = \{k|a_{ik} > b_{kj}\}$$

Note that $S(i, j) \cup T(i, j) = M$. Utilizing the commutative property of max, we observe

$$\max_{k=1,n}\{\min\{a_{ik}, b_{kj}\}\} = \max\{\max_{k\in S(i,j)}\{a_{ik}\}, \max_{k\in T(i,j)}\{b_{kj}\}\} \quad (9)$$

Let us assume sorted lists of the k-th column of A and the k-th row of B are available. The sorted lists are denoted by E_k and F_k. Let the merged list of E_k and F_k be G_k. Let H_k and L_k be the lists of ranks of elements of the k-th column of A and k-th row of B in G_k.

Then we have

$$G_k(H_k(i)) = a_{ik},\ G_k(L_k(j)) = b_{kj}$$

Let a binary vector $\mathbf{x}_{ij}$ be defined by $\mathbf{x}_{ij}[k] = 1$, if $a_{ik} \leq b_{kj}$, and 0, otherwise. Here $\mathbf{x}[k]$ is the k-th element of vector $\mathbf{x}$. The vectors $\mathbf{x}_{ij}$ and its complement $\mathbf{x}'_{ij}$ are membership vectors of $S(i, j)$ and $T(i, j)$. We express this fact by $S(i, j) = set(\mathbf{x}_{ij})$ and $T(i, j) = set(\mathbf{x}'_{ij})$.

Let pre-computed tables MAX_i^a and MAX_j^b be available, which are defined by

$$MAX_i^a(h(\mathbf{x})) = \max_{k\in set(\mathbf{x})}\{a_{ik}\} \quad (10)$$

$$MAX_j^b(h(\mathbf{x})) = \max_{k\in set(\mathbf{x})}\{b_{kj}\} \quad (11)$$

We compute MAX_i^a and MAX_j^b for all possible $\mathbf{x}$, so that we can use the table for each $\mathbf{x}_{ij}$ and $\mathbf{x}'_{ij}$. Note that those tables can be computed on A and B separately.

Also we assume that for lists of integers $H = (H_1, ..., H_m)$ and $L = (L_1, ..., L_m)$ a pre-computed table $MAP(h(H), h(L))$, defined in the following, is available.

$MAP(h(H), h(L)) = h(\mathbf{x})$, where $\mathbf{x}[k] = 1$, if $H_k \leq L_k$, and 0, otherwise.

This table will be used for $H(i) = (H_1(i), ..., H_m(i))$ and $L(j) = (L_1(j), ..., L_m(j))$ for each (i, j), where $H(i)$ and $L(j)$ are substituted for H and L.

Since $a_{ik} \leq b_{kj} \Longleftrightarrow H_k(i) \leq L_k(j)$, we can compute (9) by looking up those tables in $O(1)$ time for each i and j in the following, where MAP' is the bit-wise complement of MAP.

$$c_{ij} = max_{k=1,n}\{\min\{a_{ik}, b_{kj}\}\} = \max\{\max_{k \in set(\mathbf{x}_{ij})}\{a_{ik}\}, \max_{k \in set(\mathbf{x}'_{ij})}\{b_{kj}\}\}$$

$$= \max\{MAX_i^a(h(\mathbf{x}_{ij})), MAX_j^b(h(\mathbf{x}'_{ij}))\}$$

$$= \max\{MAX_i^a(MAP(h(H(i)), h(L(j)))), MAX_j^b(MAP'(h(H(i)), h(L(j))))\}$$

Note that the direction of scanning for packing of ranks for $h(H(i))$ and $h(L(j))$ is orthogonal to the direction of scanning for merging the lists E_k and F_k and computing H_k and L_k. See Figure 1.

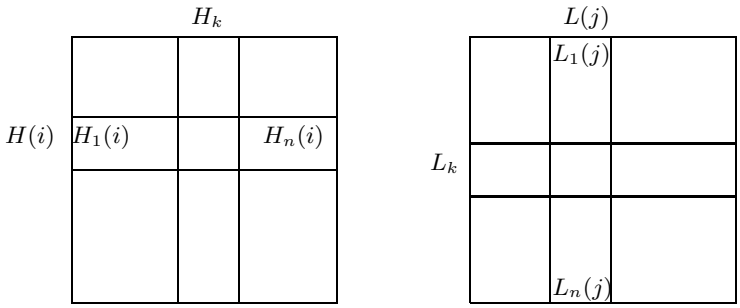

Fig. 1. H_k is column k in the left and L_k is row k in the right

$H(i)$ is row i in the left and $L(j)$ is column j in the right

Example 1. We consider $(4, 4)$-matrices. Let the i-th row of A be $\mathbf{a}_i = (12, 7, 4, 23)$ and the j-th column of B be $\mathbf{b}_j = (15, 6, 11, 20)$. Let $H(i) = (3, 2, 1, 4)$ and $L(j) = (4, 1, 3, 3)$. These numbers in $H(i)$ and $L(j)$ are determined by the relative order with other rows and columns. From this, we see $\mathbf{x}_{ij} = (1, 0, 1, 0)$ and $\mathbf{x}'_{ij} = (0, 1, 0, 1)$. $\max\{MAX_i^a(h(1, 0, 1, 0)), MAX_j^b(h(0, 1, 0, 1))\} = \max\{\max\{12, 4\},$

$\max\{6, 20\}\} = 20$. Note that $\mathbf{a}_i$ and $\mathbf{b}_j$ can contain large numbers or real numbers of any precision, whereas H_i and L_j contain only numbers between 1 and 4, and $\mathbf{x}_{ij}$ is a binary vector.

Based on the method described, we now summarise our algorithm for (max, min)-multiplication in the following. Table construction and presort are mentioned in the upper structure of the algorithm in the following sections.

ALGORITHM 3 *Compute* (max, min)*-product* $C = AB$
1. Merge E_k *and* F_k *to form* G_k *for* $k = 1, ..., m$
2. Compute H_k *and* L_k *for* $k = 1, ..., m$
3. Compute $h(H(i))$ *for* $i = 1, ..., m$
4. Compute $h(L(j))$ *for* $j = 1, ..., m$
5. Compute $c_{ij} = \max\{MAX_i^a(MAP(h(H(i)), h(L(j)))),$
$MAX_j^b(MAP'(h(H(i)), h(L(j))))\}$ *for* $i, j = 1, ..., m$

The algorithm for the average k-center is very similar. The equations (10) and (11) are replaced by

$$SUM_i^a(h(\mathbf{x})) = \Sigma_{k \in set(\mathbf{x})}\{a_{ik}\} \quad (12)$$

$$SUM_j^b(h(\mathbf{x})) = \Sigma_{k \in set(\mathbf{x})}\{b_{kj}\} \quad (13)$$

Using those mappings, we have the following algorithm.

ALGORITHM 4 *Compute* (Σ, min)*-product* $C = AB$
1. Merge E_k *and* F_k *to form* G_k *for* $k = 1, ..., m$
2. Compute H_k *and* L_k *for* $k = 1, ..., m$
3. Compute $h(H(i))$ *for* $i = 1, ..., m$
4. Compute $h(L(j))$ *for* $j = 1, ..., m$
5. Compute $c_{ij} = \max\{SUM_i^a(MAP(h(H(i)), h(L(j)))),$
$SUM_j^b(MAP'(h(H(i)), h(L(j))))\}$ *for* $i, j = 1, ..., m$

We note that both Algorithms 3 and 4 take $O(m^2)$ time.

6 How to construct the tables

In Section 3 we used h for encoding small integers. In this section, we describe h more specifically as well as how to construct tables. We choose the value of m so as to satisfy the time for table construction is manageable. For μ, we have the following two choices. When we encode ranks in $H(i)$ or $L(j)$, we set $\mu = 2m$. When we encode a binary vector of size m, we set $\mu = 2$. Let $m = O(\log n / \log\log n)$. Then the size of the table is $O(n)$ or less by choosing an appropriate value for constant factor c such that $m \leq c \log n / \log\log n$, as shown below.

To compute $MAP(\alpha, \beta) = h(x_1, x_2, ..., x_m)$ for arbitrary α and β, we decode α and β in $O(m)$ time, then compare the decoded lists of α and β, which are

regarded as H and L in the previous section, one by one to get $x_1, ..., x_m$, and finally encode $x_1, ..., x_m$, spending $O(m)$ time. We do this for all possible α and β. The possible range of α and β is up to $(2m)^m$.

The total time for computing this table is thus $O(((2m)^m)^2 m)$. Observe

$$O(((2m)^m)^2 m) = O(c^{m \log m}) = O(n), \text{ for some constant } c > 0.$$

We note that table MAP is pre-computed, that is, computed independent of the input distance matrices, whereas tables MAX_i^a and MAX_j^b are computed based on the input matrices.

Now let us compute tables MAX_i^a and MAX_j^b. To compute $MAX_i^a(\alpha)$, we decode $\alpha = h(x_1, ..., x_m)$. Then take the maximum of $\{a_{ik}\}$ such that $x_k = 1$. MAX_j^b is computed similarly. Thus the time for MAX_i^a for all $i = 1, ..., m$ and MAX_j^b for all $j = 1, ..., m$ is $O(2^m m^2)$. Let us denote the collection of $\{MAX_i^a | i = 1, ..., m\}$ and $\{MAX_j^b | j = 1, ..., m\}$ by MAX^a and MAX^b. We need to construct those tables for N^2 sub-matrices given by A_{ik} and B_{kj} in (6). The time for those N^2 collections of tables is $O(N^2 m^2 2^m) = O(n^2 2^m) = O(n^2 n^{c/\log\log n}) = O(n^{2+\epsilon})$ for any $\epsilon > 0$.

The computation of tables SUM_i^a and SUM_j^b is similar.

7 Algorithm for the whole problem and analysis

We summarise our algorithms for the absolute 2-center problem and average 2-center problem in the following. Note that we can use MAP, and other parts in common in the following two algorithms.

Algorithm 5 *Absolute 2-center*
1. Construct table MAP
2. Divide matrices A and B into A_{ij} and B_{ij} for $i, j = 1, ..., N$
3. Construct tables in MAX^a and MAX^b for A_{ij} and B_{ij} $(i, j = 1, ..., N)$
4. Sort m columns of A_{ij} and m rows of B_{ij} for $i, j = 1, ..., N$. // Presort
5. Compute $A_{ik}B_{kj}$ for $i, j = 1, ..., N$, by Algorithm 3
6. Compute $C_{ij} = \max_k\{A_{ik}B_{kj}\}$ for $i, j = 1, ..., N$
7. Compute the minimum element of matrix $C = \{C_{ij}\}$

Algorithm 6 *Average 2-center*
1. Construct table MAP
2. Divide matrices A and B into A_{ij} and B_{ij} for $i, j = 1, ..., N$
3. Construct tables in SUM^a and SUM^b for A_{ij} and B_{ij} $(i, j = 1, ..., N)$
4. Sort m columns of A_{ij} and m rows of B_{ij} for $i, j = 1, ..., N$. // Presort
5. Compute $A_{ik}B_{kj}$ for $i, j = 1, ..., N$, by Algorithm 4
6. Compute $C_{ij} = \Sigma_k A_{ik}B_{kj}$ for $i, j = 1, ..., N$
7. Compute the minimum element of matrix $C = \{C_{ij}\}$

As the above two algorithms are very similar, we analyze computing time for both algorithms together. Line 1 takes $O(n)$ time. Line 2 takes $O(n^2)$ time. Line 3 takes $O(n^{2+\epsilon})$ time. Sorting m columns of A_{ij} and m rows of B_{ij} takes $O(m^2 \log m)$ time. Thus line 4 takes $O(N^2 m^2 \log m) = O(n^2 \log\log n)$ time. Since computing $A_{ik}B_{kj}$ takes $O(m^2)$ time, line 5 takes $O(N^3 m^2) = O(n^3 \log\log n/\log n)$ time. Line 6 takes $O(n^3/m) = O(n^3 \log\log n/\log n)$ time. Line 7 takes $O(n^2)$ time. Thus in total these algorithms take $O(n^3 \log\log n/\log n)$ time.

8 When edge costs are small integers

If edge costs are small non-negative integers, the complexity for APSP becomes deeply sub-cubic, i.e., $O(n^{3-\epsilon})$ for some $\epsilon > 0$, as shown in [14], [2], [17] and [19]. It is interesting to investigate whether we can use those sub-cubic algorithms for the APSP problem for the 2-center problems. Once the APSP problem is solved, the values in matrix D^*, the all-pairs shortest distance matrix, are no-longer small integers; they can be $O(n)$, even if edge costs are all one. Thus we cannot extend the technique used for the APSP problem to the 2-center problems straight away. We can efficiently solve the absolute problem in such a case by binary search as follows.

Let us assume the APSP problem for the given graph with unit edge costs has been solved with the shortest distance from vertex i to vertex j being $d^*[i,j]$. Let the threshold value t be initialized to $n/2$. Let a Boolean matrix B be defined by its element $b[i,j]$ as follows: $b[i,j] = 1$, if $d^*[i,j] \geq t$, and 0, otherwise. Let us square B to get the matrix $C = B^2$. From the equation $c[i,j] = \Sigma_{k=1}^{n} b[i,k]b[k,j]$, we observe that $c[i,j] = 1$ if and only if $b[i,k] = 1$ and $b[k,j] = 1$ for some k. From this we derive the fact that $c_{abs} \geq t$ if and only if $C[i,j] > 0$ for some i and j. We can repeatedly halve the possible range $[\alpha, \beta]$ of c_{abs} by adjusting the threshold value of t through the binary search. The algorithm is summarized as follows.

ALGORITHM 7 *Algorithm by binary search*
$\alpha = 0$
$\beta = n$
while $\beta - \alpha > 0$
  $t = (\alpha + \beta)/2$
  $b[i,j] = 1$ *if* $d^*[i,j] > t$, *0 otherwise for* $i, j = 1, ..., n$
  Compute $C = B \times B$
  if $c[i,j] > 0$ *for some* i *and* j
    $\alpha = (\alpha + \beta)/2$
  else $\beta = (\alpha + \beta)/2$
end
$c_{abs} = \alpha$

Obviously the iteration in the while loop is done $O(\log n)$ times. Thus the total time excluding APSP becomes $O(B(n) \log n)$, where $B(n)$ is the time for multiplying (n,n) Boolean matrices. Let $M(n)$ be the time for the APSP with unit

edge costs. Then the total time including APSP becomes $O(M(n)+B(n)\log n)$, meaning that the APSP is the bottle neck, as the best known complexity for APSP with unit edge costs is $O(n^{2.575})$ and that of $B(n)$ is $O(n^{\omega})$ with $\omega = 2.376$. Thus the APSP is the bottleneck with $O(n^{2.575})$.

When edge costs are in the range of $[0, \gamma]$ for a positive integer γ, we can initialize $\beta = \gamma n$ in the above algorithm, resulting in the time of $O(B(n)(\log n + \log \gamma))$, excluding the APSP. The best time for the APSP for general γ is $O(\gamma^{1/(4-\omega)} n^{2+1/(4-\omega)})$, which is the APSP bottleneck in this case. See [19] for the APSP complexities.

9 Concluding remarks

We showed an asymptotic improvement on the time complexity of the two versions of 2-center problems; absolute 2-center and average 2-center, both of which take $O(n^3 \log\log n/\log n)$ time. As there are some algorithms for the APSP problem whose complexity is better than $O(n^3 \log\log n/\log n)$ [4], [10], etc., there may be some room for further improvement of asymptotic complexity for our problems.

If edge costs are small non-negative integers, the complexity for APSP becomes deeply sub-cubic. Once the APSP problem is solved using those sub-cubic time algorithms, the values in matrix D^*, the all-pairs shortest distance matrix, are no-longer small integers; they can be $O(n)$ or more, even if edge costs are all one. To overcome this increase of the values of matrix elements, we used the binary search idea for the absolute problem. It is not known whether we can use the same idea for the average problem.

The next step of research would be to extend the algorithm to the k-center problem. For a heuristic approach we propose to use an efficient algorithm for the 2-center algorithm repeatedly for the given graph, starting from the original graph. Then divide the set of vertices into two parts; one is the set of vertices closer to one center, and the other closer to the other center. Let G_1 and G_2 be the two sub-graphs induced from these two sets. If $c_{abs}(c_{ave})$ for G_1 is greater than that for G_2, then we solve the 2-center problem for G_1, otherwise for G_2. We can continue this process of dividing the set of vertices with the largest value of $c_{abs}(c_{ave})$ $k-1$ times for $k \geq 2$. The computing time by this approach is $O(kT(n))$ where $T(n)$ is the time for the 2-center problem, but optimality cannot be guaranteed. Thus it is our concern how close to optimal the solution is. By experiments we observe that in case of the absolute problem this approximation algorithm achieves 1.2 times the optimal value for randomly generated complete graph with $k = 4$ and $n = 64$. For practical applications, graphs are more constrained, such as planar, satisfying Euclidean distance rule, hierarchical structure, etc. It remains to be seen whether these constraints serve for better approximation ratio by this heuristic.

References

1. Aho, A. V., J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974
2. Alon, N, Galil, and O. Margalit, On the Exponent of the All Pairs Shortest 2Path Problem, Jour. Comp. Sys. Sci., vol 54, no.2 (1999)255–262,
3. Ben-Moshe, B, Bhattacharya, B. and Shi, Q. S., Efficient Algorithms for the Weighted 2-Center Problem in a Cactus Graph, ISAAC2005, LNCS 3827 (2995) 693–703
4. Chan, T., More algorithms for all pairs shortest paths, STOC 2007 (2007)590–598
5. Dobosiewicz, A more efficient algorithm for min-plus multiplication, Inter. J. Comput. Math. 32 (1990) 49-60
6. Frederickson, G. N., Parametric Search and Locating Supply Centers in Trees, WADS 1992, LNCS 915 (1992) 299-319
7. Fredman, M, New bounds on the complexity of the shortest path problem, SIAM Jour. Computing, vol. 5 (1976)83-89
8. Garey, M. R. and Johnson, D. S., Computers and Intractability, W. H. Freeman, 1979
9. Han, Y, Improved algorithms for all pairs shortest paths, Info. Proc. Lett., 91 (2004) 245-250
10. Han, Y., An $O(n^3(\log\log n/\log n)^{5/4})$ Time Algorithm for All Pairs Shortest Path, Algorithmica 51(4) (2008)428-434
11. Kariv, O and Hakimi, S. L., An Algorithmic Approach to Network Location Problems. I: The p-Centers, SIAM Jour. Appl. Math., Vol 37,No. 3 (1979)513-538
12. Kariv, O and Hakimi, S. L., An Algorithmic Approach to Network Location Problems. II: The p-Medians, SIAM Jour. Appl. Math., Vol 37,No. 3 (1979)539-560
13. Takaoka, T., A New upper bound on the complexity of the all pairs shortest path problem, Info. Proc. Lett., 43 (1992) 195-199
14. Takaoka, T, Subcubic algorithms for the all pairs shortest path problem, Algorithmica, vol. 20 (1998)309–318.
15. Takaoka, T., A Faster Algorithm for the All Pairs Shortest Path Problem and its Application, COCOON 2004, LNCS 3106 (2004) 278–280
16. Takaoka, T., An $O(n^3 \log\log n/\log n)$ time algorithm for the all pairs shortest path problem, Info. Proc. Lett., vol. 96, (2005) 154–161
17. Zwick, U, All pairs shortest paths in weighted directed graphs - exact and almost exact algorithms, 39th FOCS (1998) 310–319.
18. Zwick, U, A Slightly Improved Sub-Cubic Algorithm for the All Pairs Shortest Paths Problem, ISAAC 2004, LNCS 3341 (2004)921–932
19. Zwick, U, All pairs shortest paths using bridging sets and rectangular matrix multiplication, Jour. ACM, 49, 3 (2002) 289–317

Appendix Let us rename A_{ik} and B_{kj} in (4) by A and B. Let the difference lists, $\{a_{ir} - a_{is} | i = 1, ..., m\}$ and $\{b_{sj} - b_{rj} | j = 1, ..., m\}$, be sorted. Let $H_{rs}(i)$ and $L_{rs}(j)$ be the rank of $a_{ir} - a_{is}$ and $b_{sj} - b_{rj}$ in the list obtained by merging the above two sorted lists. Observe that

$$a_{ir} + b_{rj} \leq a_{is} + b_{sj} \Longleftrightarrow a_{ir} - a_{is} \leq b_{sj} - b_{rj} \Longleftrightarrow H_{rs}(i) \leq L_{rs}(j)$$

A sketch of the algorithm is to sort the difference lists in advance, and use the ranks of the data in a packed form in a single computer word. To determine the index k that gives the minimum to each element of the $(min, +)$-product for small matrices, we use a pre-computed table in $O(1)$ time, since the relative order of the above mentioned ranks can determine the index. An important observation is that sorting is done on data from A and B separately to minimize the time spent when A and B interact to produce the product.

In [16], it is shown that $T(m) = O(m^2(m \log m)^{1/2})$ with $m = O(\log n/(\log \log n)^3)$. Thus the time becomes $O(n^3(\log m/m)^{1/2})$. By the method of table look-up, it is shown in [16] that we can make the table in $O(n)$ time with $m = O(\log^2 n/ \log \log n)$, resulting in the total time of $O(n^3 \log \log n/ \log n)$ for the $(min, +)$-multiplication.