Faster Algorithms for Approximate Distance Oracles and All-Pairs Small Stretch Paths

Surender Baswana*
Indian Institute of Technology
Kanpur 208016, India
sbaswana@cse.iitk.ac.in

Telikepalli Kavitha
Indian Institute of Science
Bangalore 560012, India
kavitha@csa.iisc.ernet.in

Abstract

Let $G = (V, E)$ be a weighted *undirected graph with $|V| = n$ and $|E| = m$. An estimate $\hat{\delta}(u, v)$ of the distance $\delta(u, v)$ in G between $u, v \in V$ is said to be of stretch t iff $\delta(u, v) \le \hat{\delta}(u, v) \le t \cdot \delta(u, v)$. The most efficient algorithms known for computing small stretch distances in G are the* approximate distance oracles *of [16] and the three algorithms in [9] to compute all-pairs stretch t distances for $t = 2, 7/3,$ and 3. We present faster algorithms for these problems.*

For any integer $k \ge 1$, Thorup and Zwick in [16] gave an $O(kmn^{1/k})$ algorithm to construct a data structure of size $O(kn^{1+1/k})$ which, given a query $(u, v) \in V \times V$, returns in $O(k)$ time, a $2k - 1$ stretch estimate of $\delta(u, v)$. But for small values of k, the time to construct the oracle is rather high. Here we present an $O(n^2 \log n)$ algorithm to construct such a data structure of size $O(kn^{1+1/k})$ for all integers $k \ge 2$. Our query answering time is $O(k)$ for $k > 2$ and $\Theta(\log n)$ for $k = 2$. We use a new generic scheme for all-pairs approximate shortest paths for these results. This scheme also enables us to design faster algorithms for all-pairs t-stretch distances for $t = 2$ and $7/3$, and compute all-pairs almost *stretch 2 distances in $O(n^2 \log n)$ time.*

1. Introduction

The all-pairs shortest paths (APSP) problem is one of the most fundamental algorithmic graph problems. Efficient algorithms for the APSP problem are very important in several applications. However, the complexity of the fastest known algorithm for the APSP problem in a graph with m edges and n vertices with real non-negative edge weights is $O(mn + n^2 \log \log n)$ [14]. For graphs with $m = \Theta(n^2)$, this algorithm has a running time of $\Theta(n^3)$ which matches that of the old and classical algorithm of Floyd and Warshal. In fact, the best known upper bound on the worst case time complexity of this problem is $O(n^3/\log n)$ due to Chan [7], which is marginally subcubic. Surprisingly, despite the fundamental importance of the problem, there does not exist any algorithm at present that can solve APSP in truly subcubic time, i.e., $O(n^{3-\epsilon})$ for some $\epsilon > 0$. This has motivated researchers to explore ways to design efficient algorithms for the all-pairs *approximate* shortest paths (APASP) problem. In this paper we consider the problem of designing combinatorial algorithms (i.e., without using fast matrix multiplication), that are efficient both in terms of space and time required, to compute approximate shortest paths/distances between all pairs of vertices. The final goal of this research is two-fold: to achieve quadratic time bounds for all-pairs approximate distances *and* to obtain space-optimal data structures for this problem in weighted graphs.

Thorup and Zwick [16] in a remarkable result in 2001 showed that for any integer $k \ge 1$, a data structure of size $O(kn^{1+1/k})$ can be constructed that answers distance queries with a *stretch*[1] $2k - 1$. The query answering time is $O(k)$; since k is typically a small constant, this can be regarded as constant time. This data structure was called an approximate distance oracle. The space requirement of the oracle is essentially optimal, assuming a 1963 girth lower bound conjecture of Erdös [12]. However the time taken to construct the oracle is $O(kmn^{1/k})$ which is rather high when k is a small constant. We are, after all, interested in fast algorithms for small stretch distances, or equivalently, fast algorithms for small values of k and a running time of $O(m\sqrt{n})$, which could be $\Theta(n^{5/2})$ in the worst case, seems rather high for a stretch 3 oracle. In fact, there are faster algorithms known to compute all-pairs small stretch distances. Cohen and Zwick in [9] present an $O(n^2 \log n)$ algorithm to compute an $n \times n$ table that stores all-pairs

*This work was done while the author was at the Max-Planck-Institut für Informatik, Saarbrücken, Germany.

[1] An estimate $\hat{\delta}(u, v)$ of the distance $\delta(u, v)$ between $u, v \in V$ has stretch t if and only if $\delta(u, v) \le \hat{\delta}(u, v) \le t \cdot \delta(u, v)$.

stretch 3 distances. However, the space requirement in [9] is n^2 while the space required by the stretch 3 oracle is $O(n^{3/2})$. Thus the Cohen-Zwick algorithm [9] is preferred when time has to be optimized and the Thorup-Zwick algorithm [16] is preferred when space has to be optimized. Ideally, one would like an algorithm with the time requirement of the former *and* the space requirement of the latter. Thorup and Zwick in [16] ask: *is it possible to construct a stretch 3 oracle that uses only $O(n^{3/2})$ space in $\tilde{O}(n^2)$ time?*

In this paper we almost answer this question and show several other results.

1. We build a data structure of size $O(n^{3/2})$ in expected time $O(\min(n^2 \log n,\ m\sqrt{n}))$ that answers any distance query, with stretch 3, in $\Theta(\log n)$ time.

   So the only place where our algorithm falls short of combining the best features of the algorithms in [16] and [9] is in the query answering time, which was $O(1)$ in those papers.

2. For integers $k > 2$, we give an algorithm with expected running time $O(\min(n^2,\ kmn^{1/k}))$ to construct an approximate distance oracle of size $O(kn^{1+1/k})$ that answers any distance query, with stretch $(2k-1)$, in $O(k)$ time. Thus we are able to construct these oracles in quadratic time and essentially optimal space for stretch 5 and more.

The algorithms in [9] and [16] appear to be based on schemes that are quite different. This meant that it would be a nontrivial task to come up with a scheme that achieves both quadratic time and $O(n^{3/2})$ space for the stretch 3 oracle problem. We obtain our results with a new generic scheme for the APASP problem that basically captures the essence of both these schemes simultaneously.

New Tools and other new results. The new scheme $\mathcal{A}(S,k)$ has two parameters : a subset $S \subseteq V$ and an integer k. When $S = V$ and $k = \log n$, this scheme has a certain degree of similarity with the $\tilde{O}(n^2)$ algorithm of Cohen and Zwick [9] for all-pairs stretch 3 distances. However, we provide a different analysis of $\mathcal{A}(V, \log n)$ that leads to a stretch *close* to 2. The scheme $\mathcal{A}(V, \log n)$ constructs a hierarchy $S_0 = V \supseteq S_1 \supseteq \cdots \supseteq S_{\log n} = \emptyset$ and vertices in S_i (the expected number of such vertices is $n/2^i$) perform a single source shortest paths computation in a subgraph of G whose expected number of edges is $O(2^{i+1}n)$. This directly leads to a simple algorithm with expected running time $O(n^2 \log n)$ that constructs an $n \times n$ table which stores for any two vertices u and v, a distance of at most $2\delta(u,v)+$ weight of the heaviest edge on a shortest path between u and v. We call this a $(2,\omega)$-approximation.

• For our $O(n^{3/2})$ sized data structure to answer distance queries with stretch 3, we cannot afford to take $S = V$ in the scheme $\mathcal{A}(S,k)$ as we did in the $(2,\omega)$ result. However, an important property of the scheme $\mathcal{A}(S,k)$ that we show is: *Even for vertices $u \in V \setminus S$, the scheme $\mathcal{A}(S,k)$ implicitly stores 3-approximate distances from u to all those vertices in the graph that are closer to u than the vertex in S nearest to u.* Hence we take a randomly sampled subset of expected size $\sqrt{n}$ from V as S and we build a hierarchy of sets as in the $(2,\omega)$ result but with the starting set S_0 equal to our random sample S. However, this step alone does not lead to a quadratic time bound for the construction of a stretch 3 data structure. One obstacle here is to efficiently compute for each vertex u, the set $ball(u, V, S)$ (see Definition 3.1) which consists of all those vertices that are closer to u than the vertex in set S nearest to u. We present a faster algorithm for computing $balls$ which leads to an expected running time of $O(\min(n^2,\ m\sqrt{n}))$ for this problem.

• For the stretch $(2k-1)$ distance oracle for $k > 2$, we use a hierarchy $S_0 = V \supseteq S_1 \supseteq \cdots \supseteq S_{k-1} \supseteq S_k = \emptyset$ as introduced in [16]. Main computation task here is to efficiently compute the sets $ball(u, S_{i-1}, S_i)$ $\forall u \in V$ for $0 < i \leq k$. Based on our faster algorithm for computing $balls$, we achieve an expected running time of $O(\min(n^{\frac{k+i+1}{k}},\ mn^{1/k}))$ for this task provided $i < k$. For $i = k$, we show that it suffices to compute single-source shortest paths from the vertices of S_{k-1} in a sparse subgraph of G, which is obtained by using a *graph spanner*; such an idea was also used in [4, 6].

• It was shown in [16] that any stretch t distance oracle with $t < 3$ must use $\Theta(n^2)$ space on at least one input graph. Hence, in algorithms that compute all-pairs stretch t distances for $t < 3$, what we seek to optimize is the running time of the algorithm, since the space requirement is $\Theta(n^2)$. Here we present faster algorithms for computing all-pairs stretch t distances for $t = 2$ and $7/3$. The previous best algorithms for these problems (given in [9]) had running times $\tilde{O}(m^{1/2}n^{3/2})$ and $\tilde{O}(n^{7/3})$, respectively. Our new algorithms have expected running times $\tilde{O}(m\sqrt{n} + n^2)$ and $\tilde{O}(m^{2/3}n + n^2)$ for computing all-pairs stretch t distances, for $t = 2$ and $7/3$, respectively. These algorithms use an augmented scheme $\bar{\mathcal{A}}(S,k)$.

Background. During the last 10-15 years, many new combinatorial algorithms [2, 8, 1, 10, 9, 16, 3, 11] were designed for the APASP problem in order to achieve faster running times. The best results for this problem in weighted undirected graphs were the algorithms for approximate distance oracles by Thorup and Zwick in [16] and the all-pairs stretch t distance algorithms for $t = 2, 7/3$, and 3 by Cohen and Zwick in [9]. However, more results, in particu-

Stretch	Query Time	Space	Preprocessing Time	Reference
1	$O(1)$	$O(n^2)$	$O(mn + n^2 \log\log n))$	[14]
$1+\epsilon$	$O(1)$	$O(n^2)$	$\tilde{O}(n^{2.376})$	[17]
2	$O(1)$	$O(n^2)$	$\tilde{O}(n^{3/2}\sqrt{m})$	[9]
			$\tilde{O}(m\sqrt{n} + n^2)$	**this paper**
$7/3$	$O(1)$	$O(n^2)$	$\tilde{O}(n^{7/3})$	[9]
			$\tilde{O}(m^{2/3}n + n^2)$	**this paper**
3	$O(1)$	$O(n^{3/2})$	$O(m\sqrt{n})$	[16]
	$O(1)$	$O(n^2)$	$O(n^2 \log n)$	[9]
	$O(\log n)$	$O(n^{3/2})$	$O(\min(n^2 \log n, m\sqrt{n}))$	**this paper**
$2k-1$ (for $k > 2$)	$O(k)$	$O(kn^{1+1/k})$	$O(kmn^{1/k})$	[16]
			$O(\min(n^2, kmn^{1/k}))$	**this paper**

Table 1. Exact and Approximate distance oracles for weighted undirected graphs.

lar, results with *additive* error are known when the graph is unweighted. Aingworth *et al.* [1] showed a simple and elegant $\tilde{O}(n^{5/2})$ algorithm for finding all distances in unweighted undirected graphs with an additive one-sided error of at most 2. Dor *et al.* [10] improved and extended this algorithm for arbitrary additive error $2k$ for any integer $k > 0$. The algorithms in [3, 11] compute approximate distances in unweighted graphs with both multiplicative as well as additive errors simultaneously. All the algorithms for reporting approximate distances that we mentioned above, including the new results of this paper, can be easily modified to also report the corresponding approximate shortest path, and the time for doing so is proportional to the number of edges in the approximate shortest path.

Organization of the paper. The preliminaries and our scheme $\mathcal{A}(S,k)$ are given in Section 2. In Section 3 we describe our algorithms for all-pairs stretch 2 and $(2,\omega)$ distances; the stretch 7/3 result will be presented in the full version of the paper. In Section 4 we describe our $O(n^{3/2})$ sized data structure, that can be constructed in $O(n^2 \log n)$ time to report stretch 3 distances. In Section 5 we present our $O(n^2)$ algorithm to construct *balls*. In Section 6 we give the algorithm for computing $(2k-1)$-approximate distance oracles for any $k > 2$.

2. $\mathcal{A}(S,k)$: a hierarchical scheme for APASP

Preliminaries. Let $G = (V,E)$ be an undirected graph with $|V| = n$ and $|E| = m$. Let $\omega : E \to \mathbb{R}^+$ be a weight function on its edge set. For any $v \in V$, we use $E(v)$ to denote the set of edges incident on v. We use $\delta(u,v)$ to denote the distance between u and v in G. The other notations that will be used throughout the paper are the following:

For any $S \subseteq V$ and $v \in V$, let $p_S(v)$ be the vertex in the set S that is nearest to v (ties are broken arbitrarily); let $\delta(v,S)$ be the distance from v to $p_S(v)$ (define $\delta(v,S) = \infty$ if $S = \emptyset$). Given a set $S \subseteq V$, it is quite easy to compute $p_S(v)$ and $\delta(v,S)$ for all $v \in V$ in $O(m + n\log n)$ time as follows. Connect a dummy vertex to all the vertices of the set S with edges of weight zero, and run Dijkstra's algorithm starting from this vertex on the graph. We define $\mathcal{E}_S(v)$ be the set of edges incident on vertex v with weight less than $\delta(v,S)$; if $v \in S$, then $\mathcal{E}_S(v) = \emptyset$. Let $\mathcal{E}_S = \cup_{v\in V}\mathcal{E}_S(v)$. The following lemmas, which are simple to show, will be used repeatedly in the paper.

Lemma 2.1 *Given a graph $G = (V,E)$ and $S \subseteq V$, the following assertions are true.*
1. For any two vertices $u,v \in V$, if $\delta(u,v) < \delta(u,p_S(u))$, then the subgraph $(V,\mathcal{E}_S)$ preserves the exact distance between u and v.
2. The subgraph $(V,\mathcal{E}_S \cup E(p_S(u)))$ preserves the exact distance between u and $p_S(u)$.

Lemma 2.2 *If the set $S \subseteq V$ is formed by selecting each vertex independently with probability q, then the expected size of the set $\mathcal{E}_S$ is $O(n/q)$.*

The scheme $\mathcal{A}(S,k)$. Given a weighted undirected graph $G = (V,E)$, a subset $S \subseteq V$, and an integer $k > 0$, we describe in Figure 1 a hierarchical scheme $\mathcal{A}(S,k)$ for APASP. Let $p_i(v)$ denote $p_{S_i}(v)$ for $0 \le i < k$.

Lemma 2.3 bounds the time and space requirements for the scheme $\mathcal{A}(S,k)$ when the set S is formed by random sampling.

Lemma 2.3 *Given a graph $G = (V,E)$ and $k \in \mathbb{Z}^+$, let S be a set formed by selecting each vertex independently with probability $n^{-\alpha}$, for some $0 \le \alpha < 1$. The scheme $\mathcal{A}(S,k)$ can be constructed in expected $O(\min(kn^{2+1/k},\ mn^{1-\alpha}))$ time and its expected space requirement is $O(nk + n^{2-\alpha})$.*

input: $G = (V, E)$, $S \subseteq V$, and $k \in \mathbb{Z}^+$.

– Construct the hierarchy $S_0 = S \supseteq S_1 \supseteq \cdots \supseteq S_{k-1} \supseteq S_k = \emptyset$, where S_i, for $0 < i < k$ contains each element of S_{i-1}, independently, with probability $|S|^{-1/k}$.

– For $0 \le i < k$ do

1. for each $u \in V$, compute $\delta(u, S_i)$ and $p_i(u)$ (i.e., the vertex in S_i that is nearest to u).
2. for each $s \in S_i$, run Dijkstra's algorithm with source s in the subgraph $(V, \mathcal{E}_{S_{i+1}} \cup E(s))$.
   (s stores all these distances that it computes).

   Recall that $\mathcal{E}_{S_{i+1}} = \cup_{v \in V}\{$edges incident on v whose weight is strictly less than $\delta(v, S_{i+1})\}$.

Figure 1. The scheme $\mathcal{A}(S, k)$.

3. Faster algorithms for small stretch distances

3.1 All pairs $(2, \omega)$-approximate distances

Our algorithm for all pairs $(2, \omega)$-approximate distances is essentially the scheme $\mathcal{A}(S, k)$ with $S = V$ and $k = \log n$. This algorithm is based on the following important property of $\mathcal{A}(S, k)$ (proved in Theorem 3.1 below) : *For any two vertices $u, v \in S$, the scheme $\mathcal{A}(S, k)$ stores a $(2, \omega)$-approximate distance between them.*

1. Construct the scheme $\mathcal{A}(V, \log n)$ for G [using the algorithm in Figure 1] and let d be an empty $n \times n$ table.
2. For each $u, v \in V$,

   $d[u, v] \leftarrow \min_{0 \le i < k}\{\delta(u, p_i(u)) + \hat{\delta}(p_i(u), v),$
   $\delta(v, p_i(v)) + \hat{\delta}(p_i(v), u)\}$

   where $\hat{\delta}(p_i(x), y)$ denotes the distance between $p_i(x)$ and y in the subgraph $(V, \mathcal{E}_{S_{i+1}} \cup E(p_i(x)))$. This distance is retrieved from $\mathcal{A}(V, \log n)$ in $O(1)$ time.

Figure 2. Our algorithm for computing all-pairs $(2, \omega)$-approximate distances in G.

Theorem 3.1 *Let u and v be any two vertices in the set S, and let ω_{uv} be the weight of the maximum weight edge on a shortest path between u and v. Given the scheme $\mathcal{A}(S, k)$,*

$$\min_{0 \le j < k}\{\delta(u, p_j(u)) + \hat{\delta}(p_j(u), v), \delta(v, p_j(v)) + \hat{\delta}(p_j(v), u)\}$$

is at most $2\delta(u, v) + \omega_{uv}$.

Proof. Let $i < k$ be such that $u \in S_i$ but $u \notin S_{i+1}$. Note that such a unique $i < k$ must exist since $S_0 = V$ and $S_k = \emptyset$. Now consider the shortest path P_{uv} between u and v. If the entire path P_{uv}, that is, all its edges are present in $\mathcal{E}_{S_{i+1}} \cup E(u)$, then (the length of) this path will be computed by the scheme $\mathcal{A}(S, k)$ when we run Dijkstra's algorithm from u in the graph $(V, \mathcal{E}_{S_{i+1}} \cup E(u))$. Otherwise let $i+1 \le j < k$ be such that $P_{uv} \notin \mathcal{E}_{S_j}$ but $P_{uv} \in \mathcal{E}_{S_{j+1}}$. Such a j must exist since $(V, \mathcal{E}_{S_k})$ is the entire graph. Since the path $P_{uv} \notin \mathcal{E}_{S_j}$, there are one or more edges in P_{uv} that are not present in $\mathcal{E}_{S_j}$. Traversing the path P_{uv} in the direction from u to v, let (x, y) be the first such edge. Since $(x, y) \notin \mathcal{E}_{S_j}$, so as a corollary to Lemma 2.1 (item *1.*), we have

$$\delta(u, p_j(u)) \le \delta(u, y) \quad \text{and} \quad \delta(v, p_j(v)) \le \delta(v, x).$$

Since the shortest path P_{uv} has the form $u \rightsquigarrow x \rightarrow y \rightsquigarrow v$, adding the above two inequalities, we get

$$\begin{aligned} \delta(u, p_j(u)) + \delta(v, p_j(v)) &\le \delta(u, y) + \delta(x, v) \\ &= \delta(u, v) + \omega(x, y). \end{aligned}$$

Without loss of generality assume that $\delta(u, p_j(u)) \le \delta(v, p_j(v))$. Then we have

$$2\delta(u, p_j(u)) \le \delta(u, v) + \omega(x, y). \tag{1}$$

Recall that during the construction of scheme $\mathcal{A}(S, k)$, we execute Dijkstra's algorithm from $p_j(u)$ in the graph $(V, \mathcal{E}_{S_{j+1}} \cup E(p_j(u)))$. Since $\mathcal{E}_{S_j} \subseteq \mathcal{E}_{S_{j+1}}$, it follows from Lemma 2.1 (item *2.*) that the shortest path from $p_j(u)$ to u is preserved in the graph $(V, \mathcal{E}_{S_{j+1}} \cup E(p_j(u)))$. It follows from the definition of j that the shortest path between u and v is also preserved in this graph. Using these observations and applying the triangle inequality, the distance $\hat{\delta}(p_j(u), v)$ between $p_j(u)$ and v in the graph $(V, \mathcal{E}_{S_{j+1}} \cup E(p_j(u)))$ can be bounded from above as follows.

$$\hat{\delta}(p_j(u), v) \le \delta(p_j(u), u) + \delta(u, v) \tag{2}$$

Using Inequalities (1) and (2), it follows that $\delta(u, p_j(u)) + \hat{\delta}(p_j(u), v)$ is bounded from above by $2\delta(u, v) + \omega(x, y)$.

So if ω_{uv} is the weight of the heaviest edge on the shortest path between u and v, then the distance $\delta(u, p_j(u)) + \hat{\delta}(p_j(u), v)$ does not exceed $2\delta(u, v) + \omega_{uv}$.

Since we employ the scheme $\mathcal{A}(S, k)$ with $S = V$ in our algorithm, it follows from Theorem 3.1 that at the end of our algorithm in Figure 2, we have $d[u, v] \le 2\delta(u, v) + \omega_{uv}$, for all $u, v \in V$. In other words, the algorithm computes all-pairs $(2, \omega)$-approximate distances. Furthermore, since $k = \log n$, it follows from Lemma 2.3 that the expected running time of the algorithm is $O(n^2 \log n)$.

Theorem 3.2 *All-pairs $(2, \omega)$-approximate distances in G can be computed in expected $O(n^2 \log n)$ time.*

3.2 All pairs stretch 2 distances

This algorithm is based on an augmentation of our scheme $\mathcal{A}(V, k)$ (Figure 1). We first need the concepts of *balls* and *clusters* used in [15] and [16].

Definition 3.1 *For a vertex v in a graph $G = (V, E)$, and subsets X, Y of vertices, the set $ball(v, X, Y)$ of vertices is defined as : $ball(v, X, Y) = \{x \in X | \ \delta(v, x) < \delta(v, Y)\}$.*

As a simple observation, it can be noted that $ball(v, X, \emptyset)$ is the set X itself, whereas $ball(v, X, X) = \emptyset$. The following lemma about balls from [16] plays a key role in achieving a subquadratic size and subcubic preprocessing time for the approximate distance oracles in [16]. These oracles basically store distances to a hierarchy of balls around each vertex.

Lemma 3.3 *[16] Given a graph $G = (V, E)$, let the set Y be formed by picking each vertex of a set $X \subseteq V$ independently with probability q. Then*
(i) $\mathbf{E}[|ball(v, X, Y)|]$ is at most $1/q$, and
(ii) it takes expected $O(m/q)$ time to compute $\{ball(v, X, Y) | \ v \in V\}$.

Definition 3.2 *For any sets $X, Y \subseteq V$ and a vertex $x \in X$, a* cluster *$C(x, Y)$ of vertices is a set defined as follows : $C(x, Y) = \{v \in V | \ \delta(v, x) < \delta(v, Y)\}$.*

It is easy to see that $v \in C(x, Y)$ if and only if $x \in ball(v, X, Y)$, that is, balls and clusters are "inverses" of each other. The following lemma from [15] shows how to compute a set S such that the sizes of all the clusters (formed with respect to distance from S) are small.

Lemma 3.4 *[15] Given a weighted graph $G = (V, E)$ and $0 < q < 1$, we can compute a set $S \subset V$ of size $O(nq \log n)$ in expected time $O(m/q \log n)$ such that $|C(x, S)| = 1/q$ for all $x \in V$.*

In this section, we use Lemmas 3.3 and 3.4 for computing balls and clusters. In Section 5, we present a faster algorithm for computing balls; this faster algorithm is needed in our algorithms for approximate distance oracles.

We present now the augmented scheme $\bar{\mathcal{A}}(V, k)$ for a given graph G and a positive integer k.

1. Initialize the $n \times n$ matrix d as: $d[x, y] = \omega(x, y)$ if $(x, y) \in E$ and ∞ otherwise.
2. Construct the hierarchy $S_0 = V \supseteq S_1 \supseteq S_2 \cdots \supseteq S_k \supseteq S_{k+1} = \emptyset$ as described below:
   (i) First construct the hierarchy $A_0 = V \supseteq A_1 \supseteq A_2 \cdots \supseteq A_k$
   where $A_i, 0 < i \leq k$, contains each element of A_{i-1}, independently, with probability $|n|^{-1/2k}$.
   (ii) Let $S \subset V$ be a set of $O(\sqrt{n})$ vertices such that $|C(x, S)| = O(\sqrt{n} \log n), \ \forall x \in V$.
   (use Lemma 3.4 for this step with $q = 1/\sqrt{n} \log n$)
   (iii) Let $S_i \longleftarrow A_i \cup S$, for all $0 \leq i \leq k$.
3. For $i \leq k$, compute $p_i(u)$ and $\delta(u, p_i(u))$ for all $u \in V$ (recall that $p_i(v)$ denotes $p_{S_i}(v)$).
4. For each $u \in V \setminus S_k$ compute $ball(u, V, S_k)$. In addition, do the following processing :
   For each $x \in ball(u, V, S_k)$ and $0 \leq i \leq k$
   For each neighbor y of x do:
   $$d[p_i(u), y] \longleftarrow \min \begin{cases} d[p_i(u), y] \\ \delta(u, p_i(u)) + \delta(u, x) + \omega(x, y) \end{cases}$$
5. For $0 \leq i \leq k$ and for each $s \in S_i$
   – run Dijkstra's algorithm with source s in the subgraph $(V, \mathcal{E}_{S_{i+1}} \cup d[s])$.
   (Note that $d[s] = \{(s, x), \forall x \in V$ with edge weight $d[s, x].\})$

Lemma 3.5 *Let y be a vertex and P_{uy} be a shortest path from u to y. If y or the predecessor of y on the path P_{uy} belongs to $ball(u, V, S_k)$, then for any $0 \leq i \leq k$, it follows from step 4 of the above algorithm that $d[p_i(u), y] \leq \delta(u, p_i(u)) + \delta(u, y)$.*

If $u \in ball(v, V, S_k)$ or $v \in ball(u, V, S_k)$, we have the exact u-v distance computed due to step 4 of the scheme $\bar{\mathcal{A}}(V, k)$. Otherwise stretch 2 distance between u and v is stored implicitly in the scheme as shown by the following theorem, which can be proved along similar lines to Theorem 3.1.

Theorem 3.6 *Given a weighted graph $G = (V, E)$ and the augmented scheme $\bar{\mathcal{A}}(V, k)$, let u and v be any two vertices such that $u \notin ball(v, V, S_k)$ and $v \notin ball(u, V, S_k)$. Then the following assertion holds:*

$$\min_{0 \leq j \leq k} \{\delta(u, p_j(u)) + \hat{\delta}(p_j(u), v), \delta(v, p_j(v)) + \hat{\delta}(p_j(v), u)\}$$

is at most $2\delta(u, v)$, where $\hat{\delta}(p_i(x), y)$ is the distance between $p_i(x)$ and y in the subgraph $(V, \mathcal{E}_{S_{i+1}} \cup d[p_i(x)])$ as computed in step 5 of the above algorithm.

Theorem 3.6 immediately leads to our algorithm in Figure 3, that stores all-pairs stretch 2 distances in the table d.

1. Compute the scheme $\bar{\mathcal{A}}(V, \log n)$.
2. For each $u, v \in V$ do
 – If $v \in ball(u, V, S_k)$ or $u \in ball(v, V, S_k)$ then the exact u-v distance has already been computed; store this distance value in $d[u, v]$.
 – Else $d[u, v] \leftarrow \min_{0 \le i \le k}\{\delta(u, p_i(u)) + \hat{\delta}(p_i(u), v), \quad \delta(v, p_i(v)) + \hat{\delta}(p_i(v), u).\}$

Figure 3. Our algorithm for all-pairs stretch 2 distances.

The time complexity of our algorithm. The algorithm is essentially the augmented scheme $\bar{\mathcal{A}}(V, \log n)$. The formation of hierarchy $\{S_i | 0 \le i \le k+1\}$ requires expected $O(m\sqrt{n})$ time using Lemma 3.4. Step 3 requires $O(km)$ time. In step 4, the computation of balls takes expected $O(m\sqrt{n})$ time (Lemma 3.3), and the remaining task requires $\sum_{x \in V} |C(x, S_k)| deg(x)$ time, which is $\tilde{O}(m\sqrt{n})$. Step 5 requires expected $O(m\sqrt{n}\log n + n^2 \log n)$ time using Lemma 2.3. Combined together, the total expected time complexity of $\bar{\mathcal{A}}(V, \log n)$ is $\tilde{O}(m\sqrt{n} + n^2)$.

Theorem 3.7 *All-pairs stretch* 2 *distances in* G *can be computed in expected* $\tilde{O}(m\sqrt{n} + n^2)$ *time.*

4. All-pairs stretch 3 distances in quadratic time and subquadratic space

Now we present our $O(n^2 \log n)$ algorithm that computes a data structure $\mathcal{D}$ of size $O(n^{3/2})$ for all-pairs stretch 3 distances. This is based on the scheme $\mathcal{A}(S, k)$ (Figure 1). Since the size of the data structure is $o(n^2)$, we cannot afford to take $S = V$ in the scheme $\mathcal{A}(S, k)$ (as in Section 3.1). So we employ another important property of the scheme $\mathcal{A}(S, k)$, which we prove in Theorem 4.1. This says that even a small sized set S would suffice to keep all-pairs 3-approximate distances. Our algorithm to construct $\mathcal{D}$ is given below.

$\mathcal{D}$: a data structure for all-pairs 3-approximate distances

1. Form the set S by selecting each $v \in V$ independently with probability $1/\sqrt{n}$.
2. For each $v \in V \setminus S$,
 – store the distance to $p_S(v)$ and to all the vertices of the set $ball(v, V, S)$.
3. Construct the scheme $\mathcal{A}(S, \log n)$.

In order to answer an approximate distance query efficiently, the data structure $\mathcal{D}$ will keep each $ball(v, V, S)$ in a 2-*level hash table* (see [13]) so that one can determine in $O(1)$ worst case time whether any vertex u belongs to $ball(v, V, S)$, and if so, report the distance $\delta(v, u)$. Once we have the set $ball(v, V, S)$, the computation time as well as the space requirement of the corresponding hash table would be of the order of $|ball(v, V, S)|$. The algorithm for answering a 3-approximate distance query between any pair of vertices is now very easy: If $v \in ball(u, V, S)$, report the exact distance $\delta(u, v)$. Else report using $\mathcal{A}(S, \log n)$ $\min_{0 \le i < \log n}\{\delta(u, p_i(u)) + \hat{\delta}(p_i(u), v)\}$.

Thus the answering algorithm takes $O(\log n)$ time and its correctness is based on the following theorem, whose proof is along the lines of the proof of Theorem 3.1.

Theorem 4.1 *Given the scheme* $\mathcal{A}(S, k)$*, if* u *and* v *are two vertices such that* $\delta(u, v) \ge \delta(u, S)$*, then there exists some* $0 \le j < k$ *such that* $\delta(u, p_j(u)) + \hat{\delta}(p_j(u), v) \le 3\delta(u, v)$*, where* $\hat{\delta}(p_i(x), y)$ *denotes the distance between* $p_i(x)$ *and* y *in the subgraph* $(V, \mathcal{E}_{S_{i+1}} \cup E(p_i(x)))$.

Lemma 4.2 *For any* $u, v \in V$*, the distance reported by the data structure* $\mathcal{D}$ *is at most* $3\delta(u, v)$.

Proof. There are two cases depending upon whether $\delta(u, v) < \delta(u, S)$ or $\delta(u, v) \ge \delta(u, S)$. If $\delta(u, v) < \delta(u, S)$, then v must be present in $ball(u, V, S)$, and we report the exact distance $\delta(u, v)$ using the hash table associated with $ball(u, V, S)$. So let us consider the case when $\delta(u, v) \ge \delta(u, S)$. Using Theorem 4.1, it follows that the distance reported using scheme $\mathcal{A}(S, \log n)$ is at most $3\delta(u, v)$.

4.1 Preprocessing time and size of the data structure $\mathcal{D}$

Let us first analyze the space requirement of $\mathcal{D}$. The expected size of $ball(u, V, S)$ is $O(\sqrt{n})$ using Lemma 3.3. So the total expected space required for storing the hash tables for all the balls is $O(n^{3/2})$. It follows from Lemma 2.3 that the expected space requirement for the scheme $\mathcal{A}(S, \log n)$ is also $O(n^{3/2})$. Hence the expected space requirement of the data structure $\mathcal{D}$ is $O(n^{3/2})$. We shall rebuild the data structure $\mathcal{D}$ if its space requirement exceeds the expected bound by a factor larger than two. It follows from Markov's Inequality that the expected number of rebuildings of $\mathcal{D}$ required is $O(1)$.

Let us now analyze the time required to construct $\mathcal{D}$. There are two main tasks here: (i) the computation of $ball(v, V, S)$, for each v in $V \setminus S$, and (ii) the construction of the scheme $\mathcal{A}(S, \log n)$. We form S by selecting each vertex independently with probability $1/\sqrt{n}$. So it follows from Lemma 2.3 that the expected time to construct the scheme $\mathcal{A}(S, \log n)$ is $O(\min(n^2 \log n,\ m\sqrt{n}))$. Using Lemma 3.3, it follows that the computation of $ball(v, V, S), \forall v \in V$ can be performed in expected $O(m\sqrt{n})$ time. However, we need to do this in quadratic time. In the following section, we describe a modified Dijkstra's algorithm that allows us to construct all these balls in expected $O(\min(n^2,\ m\sqrt{n}))$ time (see Lemma 5.4). It thus follows that the expected time to construct the data structure $\mathcal{D}$ is $O(\min(n^2 \log n,\ m\sqrt{n}))$. Hence we can state the following theorem.

Theorem 4.3 *For any graph on n vertices and m edges, the data structure $\mathcal{D}$ occupies $O(n^{3/2})$ space and can be built in expected $O(\min(n^2 \log n,\ m\sqrt{n}))$ time.*

Combining Theorem 4.3 and Lemma 4.2, we can state the following theorem.

Theorem 4.4 *An undirected weighted graph on n vertices and m edges can be preprocessed in expected $O(\min(n^2 \log n,\ m\sqrt{n}))$ time to build a data structure of size $O(n^{3/2})$ such that it reports a 3-approximate distance for any pair of vertices in $O(\log n)$ time.*

5. Efficient computation of balls

We now present an improved algorithm for computing balls. The starting point is a close relationship between balls and clusters as observed by Thorup and Zwick [16]. It follows from Definitions 3.1 and 3.2 that a vertex $v \in V$ belongs to cluster $C(x, Y)$ if and only if $x \in ball(v, X, Y)$. Thus, as remarked earlier, balls and clusters are "inverses" of each other, and so computing the clusters $C(x, Y), \forall x \in X$ would suffice to compute $ball(v, X, Y), v \in V$. The following simple lemma makes the task of computing $C(x, Y)$ easier.

Lemma 5.1 *[16] For any $X, Y \subset V$ and $x \in X$, if a vertex $v \in V$ is present in $C(x, Y)$, then every vertex u lying on a shortest path between x and v is also present in $C(x, Y)$.*

Lemma 5.1 implies that $C(x, Y)$ appears as a truncated subtree in the shortest path tree rooted at x, and can be computed by a careful modification of Dijkstra's algorithm as shown in [16]. The algorithm in [16] is efficient since it processes only those edges whose at least one endpoint belongs to the set $C(x, Y)$. However, this is not good enough to yield a quadratic time algorithm for oracles. Here we present an algorithm which can also be viewed as a modified Dijkstra's algorithm to compute $C(x, Y)$ and it processes relatively fewer edges of the graph. This new algorithm will process an edge (u, v) only if (i) u belongs to $C(x, Y)$ *and* (ii) $\delta(x, u) + \omega(u, v) < \delta(v, Y)$, that is, the path $x \rightsquigarrow u \rightarrow v$ is a *witness* for "$v \in C(x, Y)$".

In order to avoid processing edges that do not satisfy these two conditions, our algorithm requires some preprocessing on the graph. For each neighbor v of a vertex u, let $\Delta_u(v)$ be defined as $\Delta_u(v) = \delta(v, Y) - \omega(u, v)$. We preprocess the graph as follows. For each vertex $u \in V$, we sort its adjacency list so that all its neighbors appear in non-increasing order of their Δ_u values in the list. This task will require $O(m \log n)$ time and needs to be done just once. Now we present the new algorithm *Modified_Dijkstra*(x, Y) that computes all the vertices of the cluster $C(x, Y)$.

Algorithm *Modified_Dijkstra*(x, Y), where $x \in X \setminus Y$.
– The algorithm begins with a priority queue $\mathcal{Q}$ on labels $\{\mathcal{L}[v] | v \in V\}$, with the initialization : $\mathcal{L}[x] \leftarrow 0$ and $\mathcal{L}[v] \leftarrow \infty$ for each $v \in V \setminus \{x\}$.
– `While` $\min(\mathcal{Q}) \neq \infty$ `do`

1. `extract_min`$(\mathcal{Q})$: Extract the vertex with the smallest label $\mathcal{L}[\cdot]$ from $\mathcal{Q}$. Let this vertex be u.
2. Scan the adjacency list of u until either we reach a vertex w with $\Delta_u(w) \leq \mathcal{L}[u]$ or we reach the end of the list. For each neighbor v of u with $\Delta_u(v) > \mathcal{L}[u]$ visited during this scan, decrease its label in $\mathcal{Q}$ as follows: `decrease_key`$(v, \mathcal{Q})$: $\mathcal{L}[v] \leftarrow \min\{\mathcal{L}[v],\ \mathcal{L}[u] + \omega(u, v)\}$.

We shall now analyze the correctness and efficiency of the algorithm *Modified_Dijkstra*(x, Y).

Suppose the set $C(x, Y)$ contains j vertices. Consider the sequence $\langle x(= v_1), v_2, \cdots v_j \rangle$ of vertices of $C(x, Y)$ arranged in the non-decreasing order of their distances from x. Let $\mathcal{P}^i_{xv}$ be the set of paths from x to v, each of length $< \delta(v, Y)$ such that all the intermediate vertices belong to the set $\{v_1, \cdots, v_i\}$. Let $\delta_i(x, v)$ denote the length of the shortest path from set $\mathcal{P}^i_{xv}$. We can show (using induction on i) that the assertions given below hold for the ith iteration of *Modified_Dijkstra*(x, Y) :

1. At the beginning of the iteration, the vertex v_i is extracted from $\mathcal{Q}$, and $\mathcal{L}[v_i] = \delta(x, v_i)$.
2. Only those edges (v_i, v) incident on v_i that satisfy $\delta(x, v_i) + \omega(v_i, v) < \delta(v, Y)$ are processed.
3. At the end of the iteration, for each $v \in V$, we have $\mathcal{L}[v] = \delta_i(x, v)$.

Based on the above assertions, we can conclude the following theorem.

Theorem 5.2 *The algorithm* Modified_Dijkstra(x, Y) *extracts exactly the vertices of the set $C(x, Y)$ and assigns them their correct distances from x. During this computation, it processes an edge (u, v) if and only if $x \in ball(u, X, Y)$ and the path $x \overset{P_{xu}}{\rightsquigarrow} u \rightarrow v$ is shorter than $\delta(v, Y)$, where P_{xu} is the x-u shortest path.*

Analyzing the running time of *Modified_Dijkstra*(x, Y). The algorithm basically performs two operations: `extract_min` and `decrease_key` on the priority queue $\mathcal{Q}$. If we use a Fibonacci heap, it takes $O(1)$ time for each `decrease_key` operation and $\log n$ for each `extract_min` operation. The number of `extract_min` operations is equal to the number of vertices v satisfying $x \in ball(v, X, Y)$, and the number of `decrease_key` operations is of the order of the number of edges processed. Let us assign the cost of processing an edge (u, v) to the vertex v. Theorem 5.2 tells us that if (u, v) is processed, then "the path $x \overset{P_{xu}}{\rightsquigarrow} u \rightarrow v$ is shorter than $\delta(v, Y)$". And this implies that $(u, v) \in \mathcal{E}_Y(v)$. Thus vertex v will be charged $\mathcal{E}_Y(v) + \log n$ amount of computation cost during *Modified_Dijkstra*(x, Y) if $x \in ball(v, X, Y)$, and nil otherwise.

Theorem 5.3 *Given a graph $G = (V, E)$, and sets X, Y of vertices, we can compute $ball(v, X, Y), \forall v \in V$ by running* Modified_Dijkstra*(x, Y) on each $x \in X \setminus Y$. During this computation, each vertex $v \in V$ will be charged $(\mathcal{E}_Y(v) + \log n)$ computation cost once by every vertex in $ball(v, X, Y)$.*

The following lemma is an application of Theorem 5.3, with $X = V$ and Y as a random sample of V.

Lemma 5.4 *Let S contain each vertex of the set V independently with probability $q = 1/\sqrt{n}$. Then we can compute $ball(v, V, S)$ $\forall v \in V \setminus S$ in expected $O(\min(n^2, m\sqrt{n}))$ time.*

Proof. We compute $ball(v, V, S)$ $\forall v \in V \setminus S$ by executing *Modified_Dijkstra*(x, S) on each $x \in V$. Note that we also get the distance from v to every $x \in ball(v, V, S)$ from this computation. Let us refer to Theorem 5.3 for analyzing the total computation cost (time) incurred in this task. It follows from Theorem 5.3 that $(|\mathcal{E}_S(v)| + \log n)$ computation cost will be charged to v by each vertex in $ball(v, V, S)$. So the expected cost charged to v is $\mathbf{E}[|\mathcal{E}_S(v)| \cdot |ball(v, V, S)|]$. Observe that $|\mathcal{E}_S(v)| \leq \deg(v)$ and $\mathbf{E}[|ball(v, V, S)|] = 1/q = \sqrt{n}$ using Lemma 3.3. Hence the cost charged to v is bounded by $O(\deg(v).\sqrt{n})$. However, we shall provide a tighter analysis, showing that expected cost charged to any vertex is bounded by $O(n)$ irrespective of the large degree that the vertex v may have. First, note that all the neighbors of v corresponding to the edges of $\mathcal{E}_S(v)$ are present in $ball(v, V, S)$. So $|\mathcal{E}_S(v)| \leq |ball(v, V, S)|$ always holds. Hence the expected cost charged to v is bounded from above by $\mathbf{E}[|ball(v, V, S)|^2]$. To calculate $\mathbf{E}[|ball(v, V, S)|^2]$, consider the sequence $v_1, v_2, v_3, \ldots, v_{n-1}$ of vertices of the given graph arranged in non-decreasing order of their distances from v. It can be observed that $|ball(v, V, S)| = i$ if none of $v_1, \cdots, v_i$ is selected in S, but v_{i+1} is selected in S. The probability associated with this event is $(1 - q)^i q$. Hence we can bound $\mathbf{E}[|ball(v, V, S)|^2]$ as follows:

$$\begin{aligned}
\mathbf{E}[|ball(v, V, S)|^2] &= \sum_{i=1}^{i=n-1} i^2 (1-q)^i q \\
&\leq q \sum_{i=1}^{i=\infty} i^2 (1-q)^i \\
&= q \frac{(1 + (1-q))(1-q)}{(1-(1-q))^3} \\
&\leq q \frac{2}{q^3} = \frac{2}{q^2} = 2n
\end{aligned}$$

since $q = 1/\sqrt{n}$. Note that the second equality above follows by using the identity that $\sum_{i \geq 1} i^2 x^i = \frac{(1+x)x}{(1-x)^3}$. Hence the expected cost charged to vertex v is $O(\min(n, \deg(v)\sqrt{n}))$. So the expected cost incurred in the computation of $ball(v, V, S), \forall v \in V \setminus S$ is $O(\min(n^2, m\sqrt{n}))$.

6. A $(2k - 1)$-approximate distance oracle in expected $O(n^2)$ time

In this section we describe the construction of a $(2k-1)$-approximate distance oracle for $k > 2$. It is essentially the same as the algorithm in [16] except that we use our faster algorithm for computing balls and employ a 3-spanner. A 3-spanner of a weighted graph is a subgraph such that the distance between any two vertices $u, v \in V$ in the subgraph is at most $3\delta(u, v)$. We shall use the following result concerning 3-spanners from [5].

Theorem 6.1 *[5] An undirected weighted graph on n vertices and m edges can be processed in expected $O(m)$ time to compute its 3-spanner which has $O(n^{3/2})$ edges.*

Figure 4 contains a description of the steps that we perform to compute distance estimates which we will store in our data structure.

The data structure. The final data structure $\widehat{\mathcal{D}}$ for the approximate distance oracle keeps a hash table $B(v)$ for every vertex $v \in V$. The hash table $B(v)$ stores all the vertices of the set $\cup_{i \leq k} ball(v, S_{i-1}, S_i)$ and their distances from v. Since S_i is formed by selecting each vertex of the set S_{i-1} with probability $n^{-1/k}$, it follows from Lemma 3.3 that the expected size of $ball(v, S_i, S_{i+1})$ is at most $n^{1/k}$. Hence, the hash table $B(v)$ occupies expected $O(kn^{1/k})$ space. It takes $O(1)$ time to determine whether or not a vertex x is present in $B(v)$, and, if so, the corresponding distance can also be retrieved in $O(1)$ time. We shall

1. Construct the hierarchy $S_0 = V \supseteq S_1 \supseteq \cdots \supseteq S_{k-1} \supseteq S_k = \emptyset$,
   where S_i, for $0 < i < k$ contains each element of S_{i-1}, independently, with probability $n^{-1/k}$.
2. `For` $i \leftarrow 1$ `to` $k-1$ do: compute $p_i(v)$ and $ball(v, S_{i-1}, S_i)$ for all $v \in V$.
3. Compute a 3-spanner (V, E_S).
4. Run Dijkstra's algorithm from each $x \in S_{k-1}$ in the subgraph $(V, \mathcal{E}_{S_{k-1}} \cup E(x) \cup E_S)$.

Figure 4. Computing a (2k-1)-approximate distance oracle

use $d[v, x]$ to denote the (approximate/exact) distance between v and x as stored in the hash table $B(v)$. The following lemma holds from the construction algorithm of our $(2k-1)$-approximate distance oracle.

Lemma 6.2 *If* $w \in B(v)$, *then* $d[v, w] = \begin{cases} \delta(v,w) & \textit{if } w = p_{k-1}(v) \textit{ or } w \in S_i,\ i < k-1 \\ \hat{\delta}(v,w) & \textit{otherwise} \end{cases}$

where $\hat{\delta}(v, w)$ *is the distance between* v *and* w *in the graph* $(V, \mathcal{E}_{S_{k-1}} \cup E(p_{k-1}(v)) \cup E_S)$.

Let us first analyze the computation time for the new distance oracle. The first task requires $O(n)$ time. The third task involves the computation of a 3-spanner. Theorem 6.1 shows that this task takes expected $O(m)$ time, and there are $O(n^{3/2})$ edges in the 3-spanner (V, E_S). The fourth task requires computation of shortest-path trees from each $x \in S_{k-1}$ in the graph $(V, \mathcal{E}_{S_{k-1}} \cup E(x) \cup E_S)$. Since $|E_S| = O(n^{3/2})$, the number of edges in this graph is $O(\min(m,\ n^{2-1/k}))$ using Lemma 2.2. Also note that the expected size of S_{k-1} is $O(n^{1/k})$. Hence the fourth task of the preprocessing algorithm requires expected $O(\min(mn^{1/k},\ n^2))$ time. Now let us analyze the second task that requires the computation of $p_i(v)$ and $ball(v, S_{i-1}, S_i)$. Similar to our 3-approximate distance data structure, we can compute $ball(v, S_{i-1}, S_i)\ \forall v \in V$ by performing *Modified_Dijkstra*(x, S_i) on every vertex $x \in S_{i-1}$. In the following lemma, which is basically a generalization of Lemma 5.4, we provide a tighter bound on the expected time required to compute $ball(v, S_{i-1}, S_i)\ \forall v \in V$.

Lemma 6.3 *The expected time required to compute* $ball(v, S_{i-1}, S_i)\ \forall v \in V$ *is* $O(\min(n^{\frac{k+i+1}{k}},\ mn^{1/k}))$.

The preprocessing algorithm computes $ball(v, S_{i-1}, S_i)\ \forall v \in V$ for each $1 \le i < k$. It follows from [13] that once we have the sets $ball(v, S_{i-1}, S_i), i \le k$, the construction time as well as the space requirement of the hash table $B(v)$ is of the order of $| \cup_{i \le k}\ ball(v, S_{i-1}, S_i)|$. Thus we can state the following theorem.

Theorem 6.4 *The data structure* $\widehat{\mathcal{D}}$ *can be computed in expected* $O(\min(n^2,\ kmn^{1/k}))$ *time and it occupies* $O(kn^{1+1/k})$ *space.*

We present below the query answering algorithm using the data structure $\widehat{\mathcal{D}}$.

$\mathcal{Q}(u, v)$: Reporting approximate distance between u, v
$i \leftarrow 1$
`While` $(p_{i-1}(u)) \notin B(v))$ `do`
  swap(u, v); $i \leftarrow i + 1$

`return` $d[u, p_{i-1}(u)] + d[v, p_{i-1}(u)]$.

Analyzing the query answering algorithm. The query answering procedure of the new oracle is exactly the same as the previous $(2k-1)$-approximate distance of [16]. However, its analysis is slightly different due to the slight dissimilarity in the construction. First, we provide an overview of the underlying query answering procedure before we formally state it.

Let u and v be two vertices whose approximate distance is to be reported. In order to explain the query answering procedure, we introduce the following notation.

For a pair of vertices u and v, a vertex w is said to be t-*near* to u if $\delta(u, w) \le t \cdot \delta(u, v)$. It follows from the triangle inequality that if a vertex is t-near to u then it is $(t+1)$-near to v also.

The entire query answering process is to search for a t-near vertex of u whose distance to both u and and v is known, and $t < k$. The search is performed iteratively as follows : the ith iteration begins with a vertex $w \in S_{i-1}$ which is $(i-1)$-near to u (and hence i-near to v) and its distance from u is known. It is determined whether or not its distance to v is known. We do so by checking whether or not $w \in B(v)$. Note that $w \notin B(v)$ if and only if $w \notin ball(v, S_{i-1}, S_i)$. Using Definition 3.1, therefore, it follows that $w \notin B(v)$ if and only if, for the vertex v, the vertex $p_i(v)$ is equidistant or nearer than the vertex w. But this would imply that $p_i(v)$ is also i-near to v, and note that its distance from v is known. Therefore, if the distance $\delta(v, w)$ is not known, we continue in the next iteration with vertex $w = p_i(v)$, and swap the roles of u and v. In this way we proceed gradually, searching over the sets $S_0, \ldots, S_{k-1}$. We are bound to find such a vertex within at most k iterations since the distance from S_{k-1} is known to all the vertices.

Based on the discussion above and the query answering algorithm $\mathcal{Q}(u, v)$, the following lemma holds.

Lemma 6.5 *[16] At the beginning of the ith iteration of the* `While` *loop,* $\delta(u, p_{i-1}(u)) \leq (i-1)\delta(u, v)$.

Theorem 6.6 *The algorithm $\mathcal{Q}(u, v)$ reports in $O(k)$ time a $(2k-1)$-approximate distance between u and v, for any $k > 2$.*

Proof. The query answering algorithm performs iterations of the `While` loop until the condition of the loop fails. As mentioned above, there will be at most $(k-1)$ successful iterations before the condition of the loop fails. Let the condition fail at the beginning of the ith iteration. So the following inequality follows from Lemma 6.5 when we exit the `While` loop.

$$\delta(u, p_{i-1}(u)) \leq (i-1)\delta(u, v) \tag{3}$$

We shall analyze the distance reported by the oracle on the basis of the final value of i.

Case 1 : $(i < k)$ - In this case, it follows from Lemma 6.2 that the reported distance is $\delta(u, p_{i-1}(u)) + \delta(v, p_{i-1}(u))$, which using the triangle inequality is at most $2\delta(u, p_{i-1}(u)) + \delta(u, v)$. Using Inequality (3), this distance is at most $(2i-1)\delta(u, v)$.

Case 2 : $(i = k)$ - In this case, it follows from Lemma 6.2 that the reported distance is $\delta(u, p_{k-1}(u)) + \hat{\delta}(v, p_{k-1}(u))$. There are two simple subcases here.

If $v \in ball(u, V, S_{k-1})$, then $\delta(u, v) < \delta(u, p_{k-1}(u))$. So it follows from Lemma 2.1 that $\hat{\delta}(v, u) = \delta(v, u)$ and $\hat{\delta}(u, p_{k-1}(u)) = \delta(u, p_{k-1}(u))$. Combining these equalities and using the triangle inequality, it follows that $\hat{\delta}(v, p_{k-1}(u)) \leq \delta(v, u) + \delta(u, p_{k-1}(u))$. Now applying Inequality (3) (similar to case 1), it follows that the distance reported in this subcase is at most $(2k-1)\delta(u, v)$.

If $v \notin ball(u, V, S_{k-1})$, then $\delta(u, p_{k-1}(u)) \leq \delta(u, v)$. This inequality combined with Lemma 2.1 (item 2.) implies that $\hat{\delta}(u, p_{k-1}(u)) \leq \delta(u, v)$. Since (V, E_S) is a 3-spanner, $\hat{\delta}(v, u) \leq 3\delta(v, u)$. Combining these two inequalities and using the triangle inequality, it follows that

$$\hat{\delta}(v, p_{k-1}(u)) \leq 4\delta(v, u) \tag{4}$$

Using Inequality (4) and the inequality $\delta(u, p_{k-1}(u)) \leq \delta(u, v)$, it follows that the distance reported in this subcase is at most $5\delta(u, v)$ which is obviously $\leq (2k-1)\delta(u, v)$ for $k > 2$.

Theorems 6.4 and 6.6, lead to the final theorem.

Theorem 6.7 *An undirected weighted graph on n vertices and m edges can be preprocessed in expected $O(\min(n^2, kmn^{1/k}))$ time to compute a $(2k-1)$-approximate distance oracle of size $O(kn^{1+1/k})$, for any integer $k > 2$.*

Acknowledgment. We thank Sandeep Sen for his valuable comments on the paper.

References

[1] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths(without matrix multiplication). *SIAM Journal on Computing*, 28:1167–1181, 1999.

[2] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Near-linear time construction of sparse neighborhod covers. *SIAM Journal on Computing*, 28:263–277, 1998.

[3] S. Baswana, V. Goyal, and S. Sen. All-pairs nearly 2-approximate shortest paths in $O(n^2 \operatorname{polylog} n)$ time. In *Proceedings of 22nd Annual Symposium on Theoretical Aspect of Computer Science*, volume 3404 of *LNCS*, pages 666–679. Springer, 2005.

[4] S. Baswana and S. Sen. Approximate distance oracles for unweighted graphs in $O(n^2)$ time. *ACM Transaction on Algorithms (TALG), submitted*.

[5] S. Baswana and S. Sen. A simple linear time algorithm for computing a $(2k-1)$-spanner of $O(n^{1+1/k})$ size in weighted graphs. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 384–396, 2003.

[6] S. Baswana and S. Sen. Approximate distance oracles for unweighted graphs in $\tilde{O}(n^2)$ time. In *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 271–280, 2004.

[7] T. Chan. All-pairs shortest paths with real edge weights in $O(n^3/\log n)$ time. In *Proceedings of Workshop on Algorithms and Data Structures*, volume 3608, pages 318–324, 2005.

[8] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 28:210–236, 1998.

[9] E. Cohen and U. Zwick. All-pairs small stretch paths. *Journal of Algorithms*, 38:335–353, 2001.

[10] D. Dor, S. Halperin, and U. Zwick. All pairs almost shortest paths. *Siam Journal on Computing*, 29:1740–1759, 2000.

[11] M. Elkin. Computing almost shortest paths. *ACM Transactions on Algorithms (TALG)*, 1:282–323, 2005.

[12] P. Erdős. Extremal problems in graph theory. In *Theory of Graphs and its Applications (Proc. Sympos. Smolenice,1963)*, pages 29–36, Publ. House Czechoslovak Acad. Sci., Prague, 1964.

[13] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case time. *Journal of Association of Computing Machinery*, 31:538–544, 1984.

[14] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312:47–74, 2004.

[15] M. Thorup and U. Zwick. Compact routing schemes. In *Proceedings of 13th ACM Symposium on Parallel Algorithms and Architecture*, pages 1–10, 2001.

[16] M. Thorup and U. Zwick. Approximate distance oracles. *Journal of Association of Computing Machinery*, 52:1–24, 2005.

[17] U. Zwick. All-pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of Association of Computing Machinery*, 49:289–317, 2002.