On bounded leg shortest paths problems

Liam Roditty* Michael Segal†

Abstract

Let V be a set of points in a d-dimensional l_p-metric space. Let $s, t \in V$ and let L be a real number. An L-bounded leg path from s to t is an ordered set of points which connects s to t such that the leg between any two consecutive points in the set has length of at most L. The minimal path among all these paths is the L-bounded leg shortest path from s to t. In the s-t Bounded Leg Shortest Path (stBLSP) problem we are given two points s and t and a real number L, and are required to compute an L-bounded leg shortest path from s to t. In the All-Pairs Bounded Leg Shortest Path (apBLSP) problem we are required to build a data structure that, given any two query points from V and a real number L, outputs the length of the L-bounded leg shortest path (a distance query) or the path itself (a path query). In this paper we obtain the following results:

1. An algorithm for the apBLSP problem in any l_p-metric which, for any fixed $\varepsilon > 0$, computes in $O(n^3(\log^3 n + \log^2 n \cdot \varepsilon^{-d}))$ time a data structure which approximates any bounded leg shortest path within a multiplicative error of $(1+\varepsilon)$. It requires $O(n^2 \log n)$ space and distance queries are answered in $O(\log \log n)$ time.
2. An algorithm for the stBLSP problem that, given $s, t \in V$ and a real number L, computes in $O(n \cdot \text{polylog}(n))$ the exact L-bounded shortest path from s to t. This algorithm works in l_1 and l_∞ metrics. In the Euclidean metric we also obtain an exact algorithm but with a running time of $O(n^{4/3+\varepsilon})$, for any $\varepsilon > 0$.
3. For any weighted directed graph we give a data structure of size $O(n^{2.5} \log n)$ which is capable of answering path queries with a multiplicative error of $(1+\varepsilon)$ in $O(\log \log n + \ell)$ time, where ℓ is the length of the reported path.

Our results improve upon the results given by Bose et al. [5]. Our algorithms incorporate several new ideas along with an interesting observation made on geometric spanners, which is of independent interest.

*School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. E-mail: liamr@cs.tau.ac.il

†Communication Systems Engineering Department, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel. Partially supported by REMON (4G) consortium. E-mail: segal@cse.bgu.ac.il.

1 Introduction

The All Pairs Shortest Paths (APSP) problem is one of the most fundamental algorithmic graph problems. The fastest algorithm for solving the problem in real-weighted directed graphs is due to Pettie [17] and has a running time of $O(mn + n^2 \log\log n)$. It improves the long-standing bound of $O(mn + n^2 \log n)$. Here m and n are the number of edges and vertices, respectively. If fast matrix multiplication can be used then a faster algorithm is available due to Zwick [20]. For undirected graphs with nonnegative integer edge weights, a running time of $O(mn)$ can be obtained by running a single source shortest paths algorithm of Thorup [18] from each vertex of the graph.

Consider now the following interesting problem which is a natural generalization of the classical APSP. Let the maximal leg of a path be its heaviest edge. Suppose now that there are many different types of entities that would like to move from one point to another on a shortest path. The entities are divided into types according to the maximal leg that each of them allows. In such a case, given two points s and t, each entity may have a different shortest path from s to t. Therefore, there may be as many as $\binom{n}{2}$ different paths between s and t. We now give the formal definition of the problem in geometric settings.

Let V be a set of points in a d-dimensional l_p-metric space. Let $s, t \in V$ and let L be a real number. An L-bounded leg path from s to t is an ordered set of points which connects s to t such that the leg between any two consecutive points in the set is at most L. The minimal path among all these paths is the L-bounded leg shortest path from s to t. In other words, the L-bounded leg shortest path from s to t is the shortest path from s to t in the graph G, where the weight of (x, y) is the l_p-distance between x and y or infinity if this distance exceeds L.

In the s-t Bounded Leg Shortest Path (stBLSP) problem we are given two points s and t from V and a real number L, and are required to compute an L-bounded leg shortest path from s to t. In the All-Pairs Bounded Leg Shortest Path (apBLSP) problem we are required to build a data structure that, given any two query points from V and a real number L, outputs the length of the L-bounded leg shortest path (a distance query) or the path itself (a path query).

A natural motivation for the problem comes from transportation networks. Suppose we wish to drive from one depot station to another (among n others) and our car is capable of carrying enough fuel only for L miles. In such a case we must drive on a route whose maximal leg is L. Naturally, we would like to take the shortest possible route among all routes with a maximal leg L.

The distance from a vertex u to a vertex v in a graph G is denoted by $\delta_G(u, v)$. We say that an estimate $\hat{\delta}_G(u, v)$ of the distance $\delta_G(u, v)$ is of *stretch* c if and only if $\delta_G(u, v) \leq \hat{\delta}_G(u, v) \leq c \cdot \delta_G(u, v)$. We are especially interested in obtaining stretch $1 + \varepsilon$ estimates, for an arbitrary small $\varepsilon > 0$, as they are, in most cases, as good as exact distances. (In particular, for distances up to $1/\varepsilon$, such estimated distances are exact.)

Previous results

The apBLSP problem was considered first by Bose et al. [5]. According to their terminology, the problem has been called "the geometric bottleneck shortest path problem"; however, bottleneck shortest path often means the shortest path whose maximal edge is minimal, thus we prefer to use the term "bounded leg". Given two vertices and a real number L, the data structure proposed in [5] returns a $(1+\varepsilon)$-approximation of the length of the shortest L-bounded leg path. It requires $O(n^2 \log n)$ space to answer a distance query and $O(n^3 \log n)$ space to answer a path query. A distance query takes $O(\log n)$ time and a path query takes $O(\ell + \log n)$ time, where ℓ is the number of edges on the reported path. The time required for the construction of Bose et al. data structure is $O(n^5)$. In addition, for planar graphs, Bose et al. [5] presented

a data structure of size $O(n^{2.5})$ which is able to answer path queries. However, the query time deteriorates to $O(\sqrt{n}+\ell)$.

The problem of stBLSP was not considered by the authors of [5]; however, they did consider the following restricted problem. Given two points and a real number L decide whether or not there exists an L-bounded leg path between them. The running time of their algorithm for this problem is $O(n \log n)$. To the best of our knowledge, the stBLSP problem has not been considered before.

Our contribution

We present a new algorithm for the apBLSP problem that improves upon the result of Bose et al. [5]. For a given set V of n points in a d-dimensional l_p-metric space[1] we show that a data structure which approximates any bounded leg shortest path within a stretch of $(1+\varepsilon)$ can be computed in $O(n^3(\log^3 n + \log^2 n \cdot \varepsilon^{-d}))$ time. Note that up to a poly-logarithmic factor it is essentially the time needed for only one all-pairs shortest paths computation. It requires $O(n^2 \log n)$ space and distance queries are answered in $O(\log \log n)$ time. Our algorithm is obtained by combining several new ideas, one of which is careful examination and analysis, which may be interesting in their own right, of distance information which is part of any geometric $(1+\varepsilon)$-spanner. In addition, an interested reader may also ask the following question: "If we allow stretch in the distances, then why not also allow stretch in the leg?". The answer is quite simple. Allowing a tiny increase in the leg's length may lead to significant decrease L-bounded leg shortest path.

We now turn our attention to the stBLSP problem. For the metrics l_1 and l_∞ we present an $\tilde{O}(n)$ algorithm which returns the exact L-bounded leg shortest path from s to t. For the Euclidean metric we present an algorithm with a running time of $O(n^{4/3+\varepsilon})$ which also returns the exact L-bounded leg shortest path from s to t. Both algorithms are based on a technique developed by Eppstein [12] to maintain the bichromatic closest pair, which was later used to compute shortest paths by Chan and Efrat in [8]. Furthermore, we consider another variant in which we are given two points s and t and a real number L, and are required to produce an L-bounded leg shortest path in terms of the number of edges from s to t; i.e., to find a path with a minimal number of edges that connects s and t, such that the weight of each edge on the path is at most L. We call this variant the s-t Bounded Unit Leg Shortest Path (stBULSP) problem. We show a fast algorithm for the rectilinear stBULSP problem and also propose a linear space solution for the Euclidean stBULSP problem. Recently, we have learned that using a fully dynamic randomized data structure proposed by Chan [7] leads to near linear expected runtime algorithm for Euclidean stBULSP problem. The motivation for this problem rises from wireless multicast transmissions, where one wishes to keep the energy of each transmission at a low level, and, simultaneously, reach the destination in a shortest path.

We end the paper by showing how to reduce the space usage when path queries are required. We reduce the space from $O(n^3 \log n)$ to $O(n^{2.5} \log n)$ without increasing neither the query time nor the approximation stretch. This result extends the result of Bose et al. [5] from planar graphs to general graphs. We also reduce the query time from $O(\sqrt{n}+\ell)$ to $O(\ell + \log \log n)$.

The rest of this paper is organized as follows. In the next section we describe our new algorithm for computing the apBLSP. In Section 3 we describe our algorithms for the stBLSP problem. Solutions for the stBULSP problem are given in Section 4. In Section 5 we show how to reduce the space usage.

[1] The l_p metric, for any fixed p, of a vector $z = (z_1, z_2, \cdots, z_d)$ in $\Re^d$ is given by $||z||_p = (|z_1|^p + |z_2|^p + \cdots + |z_d|^p)^{\frac{1}{p}}$; if $p = \infty$, then $||z||_\infty = \max(|z_1|, |z_2|, \cdots, |z_d|)$.

2 The all-pairs bounded leg shortest path problem

In this section we start by reviewing and simplifying some of the ideas used by Bose et al. [5]. We then proceed by showing how to build in $O(n^3(\log^3 n + \log^2 n \cdot \varepsilon^{-d}))$ time a data structure which supports distance queries for the apBLSP problem. Our algorithm applies to any complete undirected graph whose edge weights are distances under any l_p-norm metric space.

Let $G = (V, E)$ be a complete weighted undirected graph. Let $w_1, w_2, \ldots, w_{N=\binom{n}{2}}$ be the weights of its edges in increasing order[2]. Let $w(e)$ be the weight of e and let $G^i = (V, E^{[1,i]})$, where $E^{[x,y]} = \{e \mid w_x \leq w(e) \leq w_y\}$. We denote by $\delta_{G^i}(u, v)$ the length of the shortest path from u to v in G^i and by $\delta_G(u, v)$ the length of the shortest path from u to v in G. Similarly, we denote with $h_{G^i}(u, v)$ the number of edges on the shortest path from u to v in G^i and by $h_G(u, v)$ the number of edges on the shortest path from u to v in G.

We now examine the graph sequence $G^1, G^2, \ldots, G^{N=\binom{n}{2}}$ more carefully. Let $c(u, v)$ be the smallest index of the graph for which u and v are connected. This implies that any shortest path from u to v must include an edge of weight at least $w_{c(u,v)}$, thus $\delta_G(u, v) \geq w_{c(u,v)}$. Any path can have at most $n - 1$ edges thus $\delta_{G^{c(u,v)}}(u, v) \leq (n - 1)w_{c(u,v)}$. By combining these two inequalities we obtain that $\delta_{G^{c(u,v)}}(u, v) \leq (n-1)\delta_G(u, v)$. Bose et al. [5] showed that it is possible to use only $O(n^2 \log n)$ space to produce a $(1+\varepsilon)$-approximated L-bounded leg shortest path for every pair of vertices and any real number L. Given two vertices u and v the exact answer to any possible query is one of the values:

$$\delta_{G^N}(u, v) \leq \delta_{G^{N-1}}(u, v) \leq \delta_{G^{N-2}}(u, v) \leq \cdots \leq \delta_{G^{c(u,v)}}(u, v),$$

where $\delta_{G^N}(u, v) = \delta_G(u, v)$ and $\delta_{G^{c(u,v)}}(u, v) \leq (n - 1)\delta_G(u, v)$.

However, to obtain a space efficient data structure we need to choose a small subset of these values. This subset must provide a $(1 + \varepsilon)$-approximation to any possible query. This is done in the following manner. Let j be an integer such that $0 \leq j \leq \lceil \log_{1+\varepsilon}(n - 1) \rceil$. For every j, $\mathcal{I}^j(u, v)$ is the set of indices of the graphs G^i for which $\delta_{G^i}(u, v)$ satisfies $(1 + \varepsilon)^j \delta_G(u, v) \leq \delta_{G^i}(u, v) < (1 + \varepsilon)^{j+1}\delta_G(u, v)$. Note that the set $\mathcal{I}^j(u, v)$ may be empty.

Let $I^j(u, v)$ be the minimal index in $\mathcal{I}^j(u, v)$. If $\mathcal{I}^j(u, v) = \phi$ then $I^j(u, v)$ is not defined. Let $\mathcal{I}(u, v)$ be the set of all minimal indices, i.e., $\mathcal{I}(u, v) = \{I^j(u, v) \mid 0 \leq j \leq \lceil \log_{1+\varepsilon}(n - 1) \rceil\}$. For every $i \in \mathcal{I}^j(u, v)$ we use the distance from u to v in the graph $G^{I^j(u,v)}$ to produce an approximation for $\delta_{G^i}(u, v)$. Note, that it is legal to use the graph $G^{I^j(u,v)}$ to approximate a distance in G^i since $G^{I^j(u,v)}$ is a subgraph of G^i and the weight of its edges is at most w_i. In the next lemma, which is a simplified version of an argument given in [5], we prove that the stretch of the approximated path is at most $1 + \varepsilon$.

Lemma 1 *Let $j \in [0, \lceil \log_{1+\varepsilon}(n - 1) \rceil]$. If $i \in \mathcal{I}^j(u, v)$ then $\delta_{G^{I^j(u,v)}}(u, v) \leq (1 + \varepsilon)\delta_{G^i}(u, v)$.*

Proof: The set $\mathcal{I}^j(u, v)$ contains both i and $I^j(u, v)$. This implies that $\delta_{G^{I^j(u,v)}}(u, v) < (1+\varepsilon)^{j+1}\delta_G(u, v)$ and that $(1 + \varepsilon)^j \delta_G(u, v) \leq \delta_{G^i}(u, v)$. Altogether we obtain that $\delta_{G^{I^j(u,v)}}(u, v) \leq (1 + \varepsilon)\delta_{G^i}(u, v)$ □

For every pair of vertices $u, v \in V$ and for every $i \in \mathcal{I}(u, v)$ the values i, $\delta_{G^i}(u, v)$ and w_i are saved. The total number of values saved with each pair (u, v) is $O(\log n)$. Thus, the size of the data structure is $O(n^2 \log n)$. Bose et al. [5] described how to build the above data structure in $O(n^5)$ time. Their distance query time is $O(\log n)$. In order to answer path queries Bose et al. [5] save with every pair of vertices $u, v \in V$ and $i \in \mathcal{I}(u, v)$ a path of length $\delta_{G^i}(u, v)$, increasing the size of the data structure to $O(n^3 \log n)$.

[2]For the simplicity of the representation we assume that all the weights are distinct; however, this assumption can be removed easily.

```
algorithm apBLSP(G, ε)

for every pair of vertices u, v ∈ V do
  for j ← 1 to ⌈log_{1+ε} n⌉
    X ← δ_G(u, v)(1 + ε)^j
    I^j(u, v) = Search(u, v, X, 0, N)
```

```
algorithm Search(u, v, X, Left, Right)

if Right ≤ Left return Left
m ← ⌊(Left+Right)/2⌋
if (X > SP(u, v, w_m))
  return Search(u, v, X, Left, m − 1)
else
  return Search(u, v, X, m + 1, Right)
```

Figure 1: Computing all-pairs bounded leg shortest paths using binary search

As a first step we show how to reduce the query time to $O(\log \log n)$. Given a query on $u, v \in V$ and a real number L, Bose et al. [5] search for i such that $w_i \leq L < w_{i+1}$. They then do a binary search to find the largest $I^j(u,v) \in \mathcal{I}(u,v)$ which satisfies $I^j(u,v) \leq i$. In order to reduce the query time, with each index from $\mathcal{I}(u,v)$ we also save its weight. Given L we perform a binary search to find the largest weight w which satisfies $w \leq L$. The index associated with this weight is the desired index which we use to produce the approximation. As $|\mathcal{I}(u,v)| = O(\log n)$, this search takes only $O(\log \log n)$ time.

A trivial way to build the above data structure is by doing $N = \binom{n}{2}$ computations of APSP where the i-th computation is performed on the graph G^i. This gives a running time of $\tilde{O}(n^5)$, as Bose et al. [5] presented. Once the distance matrix of every graph in the sequence $G^1, G^2, \ldots, G^N$ is known, finding $\mathcal{I}(u,v)$ for every pair of vertices u and v, is trivial. We can accelerate the running time to $O(n^4)$ using a fairly standard technique. Note that the only difference between G^i and G^{i+1} is a single edge. Let this edge be (u,v). We compute the shortest paths tree of u and v in G^i. We then check for every pair of vertices $x, y \in V$ whether the value $\delta_{G^i}(x,u) + w(u,v) + \delta_{G^i}(v,y)$ is smaller than $\delta_{G^i}(x,y)$. If so, we update the distance matrix of G^{i+1} accordingly. Note that this approach will work also for weighted directed graphs.

In what follows we show how to build a $(1+\varepsilon)$ apBLSP data structure in time $O(n^3(\log^3 n + \log^2 n \cdot \varepsilon^{-d}))$ for a set of points V in a d-dimensional l_p metric space. This running time equals to a poly-logarithmic number of APSP computations.

Our improved algorithm relies on the following observation. Let $u, v \in V$ and let $\mathcal{I}(u,v)$ be the set of minimal indices we wish to find. Let $I^j(u,v) \in \mathcal{I}(u,v)$. In order to find $I^j(u,v)$ we do not need to find the set $\mathcal{I}^j(u,v)$: by the choice of $I^j(u,v)$ we know that $\delta_{G^{I^j(u,v)}}(u,v) < (1+\varepsilon)^{j+1}\delta_G(u,v)$ and that $\delta_{G^{I^j(u,v)-1}}(u,v) \geq (1+\varepsilon)^{j+1}\delta_G(u,v)$. We use the value $(1+\varepsilon)^{j+1}\delta_G(u,v)$, which we can compute in advance, to search for the index $I^j(u,v)$.

A straightforward approach to find $I^j(u,v)$ is a binary search on the graph sequence $G^1, G^2, \ldots, G^N$. We start from the middle and compute a shortest paths tree for u. We check if $(1+\varepsilon)^{j+1}\delta_G(u,v)$ is larger than $\delta_{G^{\frac{1}{2}N}}(u,v)$, and if so, we proceed to the graph $G^{\frac{1}{4}N}$; otherwise we proceed to the graph $G^{\frac{3}{4}N}$. The cost of computing $\delta_{G^{\frac{1}{2}N}}(u,v)$ is $O(n^2)$. After $O(\log n)$ steps we find the value of $I^j(u,v)$. The total cost of this search is $O(n^2 \log n)$. We need to repeat this process for every j in the range $[0, \lceil \log_{1+\varepsilon}(n-1) \rceil]$ and for every pair of vertices. The total running time will be $O((n^4 \log^2 n)/\varepsilon)$ which is roughly the same running time as the running time of the simple improvement presented above. The apBLSP algorithm based on binary search is given in Figure 1. In order to obtain a running time of $O(n^3(\log^3 n + \log^2 n \cdot \varepsilon^{-d}))$ more involved ideas are needed. Using these ideas we reduce the time needed to find each of the $O(n^2 \log n)$ values from $O(n^2 \log n)$ to $O(n \log^2 n + n \cdot \varepsilon^{-d} \log n)$. The main tool that we use is a sparse representation of the distances with only $O(n \cdot \varepsilon^{-d})$ edges. Based on this representation, we compute $(1+\varepsilon)$-approximated shortest paths in $O(n \log^2 n + n \cdot \varepsilon^{-d} \log n)$ time.

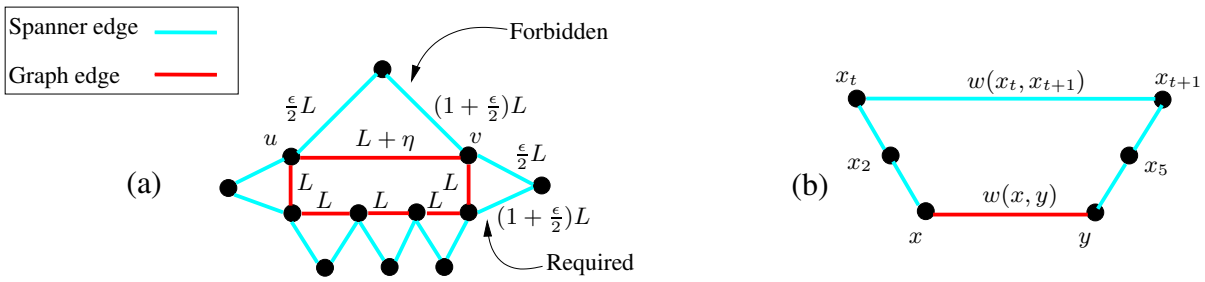

Figure 2: (a) Two edges of weight $(1+\frac{\varepsilon}{2})L$ one is required and the other is forbidden (b) The graph edge (x,y) which is approximated by the path $x_1, \cdots, x_\ell$.

The sparse representation, mentioned above, is obtained by using a *spanner*. A $(1+\varepsilon)$-spanner is a sparse subgraph of a complete undirected graph whose vertices are points in a d-dimensional l_p-metric space with only $O(\varepsilon^{-d}n)$ edges. Beside being sparse the spanner also preserves the distances up to a multiplicative error of $(1+\varepsilon)$. Vaidya [19] showed how to construct such a spanner in $O(n\log n + \varepsilon^{-d}\log(1/\varepsilon)n)$ time for any l_p-metric space (for more details on spanners see Eppstein [13]) . We denote the spanner by $H=(V,F)$, where F is its set of edges and V is the original set of points. Let $H^{[x,y]}=(V,F^{[x,y]})$ be the subgraph of the spanner H with $F^{[x,y]}=F\cap E^{[x,y]}$.

In the next lemma we give a somewhat surprising existence proof of a good approximation of any bounded leg shortest path in spanners. This lemma will be used later on by our algorithm and is crucial for its correctness. We note that this is the only place in which we use the fact that the distances are l_p-distances.

Lemma 2 *Let $u,v\in V$ and let L be a real number. Let b and c be indices which satisfy $w_b \le L < w_{b+1}$ and $w_c \le L(1+\varepsilon) < w_{c+1}$ (see Figure 3(a)). Let $H=(V,F)$ be a $(1+\varepsilon)$-spanner of G. There exists a path $u=u_1,\cdots,u_\ell=v$ in $H^{[1,c]}$ such that $\delta_{G^b}(u,v) \le \sum_{i=1}^{\ell-1} w(u_i,u_{i+1}) \le (1+\varepsilon)\delta_{G^b}(u,v)$.*

Proof: Let $u=x_1,x_2,\cdots,x_t=v$ be the vertices on the shortest path from u to v in G^b. Every edge on this path represents a distance between its endpoints. This distance must be approximated by the spanner H. Let $x_i=u_{i_1},\cdots,u_{\ell(i)}=x_{i+1}$ be the shortest path in H from x_i to x_{i+1}, where $\ell(i)$ is the number of vertices on this path. From the triangle inequality we get that $w(x_i,x_{i+1}) \le \sum_{j=i_1}^{\ell(i)-1} w(u_j,u_{j+1})$. From the fact that H is a spanner we get that $\sum_{j=i_1}^{\ell(i)-1} w(u_j,u_{j+1}) \le (1+\varepsilon)w(x_i,x_{i+1})$. Now looking at the concatenation of the paths that approximate the edges (x_i,x_{i+1}) we get:

$$\delta_{G^b}(u,v)=\sum_{i=1}^{t-1} w(x_i,x_{i+1}) \le \sum_{i=1}^{t-1}\sum_{j=i_1}^{\ell(i)-1} w(u_j,u_{j+1}) \le (1+\varepsilon)\sum_{i=1}^{t-1} w(x_i,x_{i+1}) = (1+\varepsilon)\delta_{G^b}(u,v)\ .$$

Moreover, for every edge (x_i,x_{i+1}), the heaviest edge on the path that approximates it in H is of weight at most $(1+\varepsilon)w(x_i,x_{i+1})$, otherwise it will not have a stretch of $(1+\varepsilon)$. Thus, the concatenated path is in $H^{[1,c]}$. □

In Lemma 2 we showed that given $u,v\in V$ and a real number L there exists a $(1+\varepsilon)$-approximated path in $H^{[1,c]}$. However, the existence of such a path is not enough - we must be able to find it. There might be many paths from u to v in the spanner. Moreover, there might be spanner edges in $F^{[b+1,c]}$ which create a path which is much shorter than the shortest L bounded leg shortest path (which can only use edges from $E^{[1,b]}$). Our target is to build a sparse graph based on the spanner which includes edges from $F^{[b+1,c]}$ only if the path between their endpoints in G^b is of length at most $(1+\epsilon)L$. The weight of an edge from $F^{[b+1,c]}$

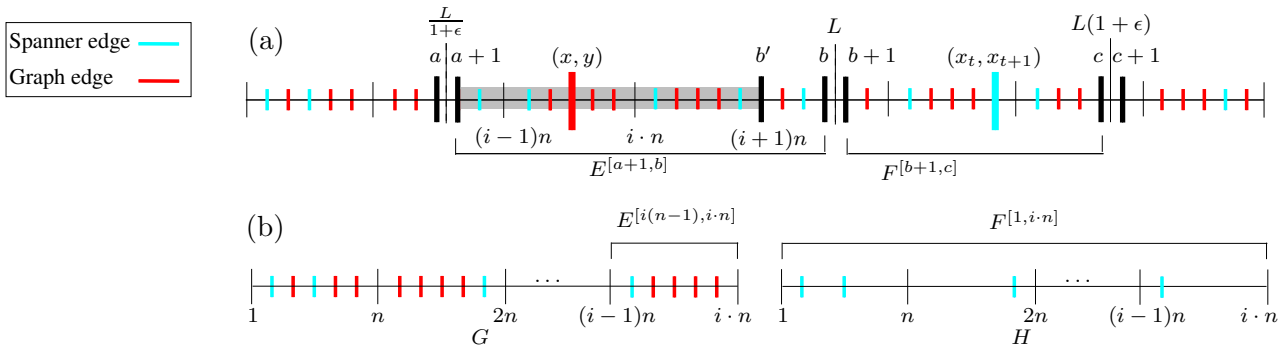

Figure 3: (a) The indices a,b and c and edges to be classified (b) The edges used to compute M_i.

is strictly larger than L, thus, the shortest path in this special constructed graph can be shorter than the path it approximates only by a factor of $1+\epsilon$.

For example, let the distance between u and v be $L+\eta$, where η is very small, and let the L-bounded leg shortest path from u to v be composed from 5 edges of weight L. The spanner approximates every distance, thus, it must have a path from u to v of length at most $(1+\varepsilon)(L+\eta)$. It might be that the heaviest edge on this path will be of weight at most $(1+\varepsilon)L$. In such a case the shortest path from u to v in $H^{[1,c]}$ is of length at most $(1+\varepsilon)(L+\eta)$. This situation is depicted in Figure 2(a). In the picture there are two edges both of weight $(1+\frac{\varepsilon}{2})L$; one is required, as it is used to approximate the L-bounded leg shortest path from u to v whereas the other is forbidden, as it creates a path which is shorter than the L-bounded leg shortest path from u to v.

We dedicate the rest of this section to explain how to distinguish between forbidden edges and those edges which are required for the approximation. While doing this process of classifying edges we create (based on the spanner H) a sparse graph with only $O(n\cdot\varepsilon^{-d})$ edges. By using this graph we obtain in $O(n\cdot\varepsilon^{-d}+n\log n)$ time a $(1+\varepsilon)$-approximation of the L-bounded leg shortest path from u to v. The following two corollaries which stem from Lemma 2 are crucial for the edge classification.

Corollary 3 *Let a and b be indices which satisfy $w_a \le \frac{L}{1+\varepsilon} < w_{a+1}$ and $w_b \le L < w_{b+1}$ (see Figure 3(a)). Every edge from $E^{[1,a]}$ is approximated by a path in $H^{[1,b]}$.*

Proof: Apply Lemma 2 with $\frac{L}{1+\varepsilon}$ instead of L. □

Corollary 4 *Let L be a real number. Let a,b and c be indices which satisfy $w_a \le \frac{L}{1+\varepsilon} < w_{a+1}$, $w_b \le L < w_{b+1}$ and $w_c \le L(1+\varepsilon) < w_{c+1}$. Let $(x,y) \in E^{[a+1,b]}$ and let $\varepsilon \le 1/3$. Every path in $H^{[1,c]}$ which approximates (x,y) has at most one edge of weight w_{a+1} or more.*

Proof: From Lemma 2 we know that there must be a path $x = x_1, x_2, \cdots, x_\ell = y$ in $H^{[1,c]}$ which approximates the distance between x and y. Suppose that there are two edges of weight at least w_{a+1}. The total length of the path will be at least $\frac{2w_b}{1+\varepsilon}$. We know that $w(x,y) \le w_b$, thus the stretch of the path is at least $2/(1+\varepsilon)$, and for $\varepsilon \le 1/3$ this is larger than $(1+\varepsilon)$. We conclude that there is at most one edge whose weight is w_{a+1} or more on this path. □

From Corollary 3 we know that the portions of the L-bounded leg shortest path from u to v which use edges from the set $E^{[1,a]}$ are approximated by $H^{[1,b]}$. Thus, our problem is only with those edges on the path that belong to the set $E^{[a+1,b]}$. Paths in $H^{[1,c]}$ which approximate these edges may require the use of

```
algorithm apBLSP(G, ε)
ε' ← ε/3
H(V,E) ← Spanner(G, ε'/3)
for i ← 1 to N/n
  M_i ← min(M_{i-1}, APSP(G(V, E^{[(i-1)n,i·n]} ∪ F^{[1,i·n]}))
for every pair of vertices u, v ∈ V do
  for j ← 1 to ⌈log_{1+ε'} n⌉
    X ← δ_G(u,v)(1+ε')^j
    I^j(u,v) = Search(u,v,X,0,N)
```

```
algorithm SP(u, v, w_b)
F' ← F^{[1,b]}
if (b − a ≤ n) then
  F' ← F' ∪ E^{[a,b]}
else
  b' ← ⌊b/n⌋n
  F' ← F' ∪ E^{[b',b]}
  for every (p,q) ∈ F^{[b+1,c]} do
    if M_{b'}[p,q] ≤ (1+ε')w_b then
      F' ← F' ∪ {(p,q)}
return (1+ε'/3)δ_{H'}(u,v)
```

Figure 4: Computing all-pairs bounded leg shortest paths

spanner edges from the set $F^{[b+1,c]}$. However, as we already explained before, some of the edges of $F^{[b+1,c]}$ may create a path which is shorter than the L-bounded leg shortest path from u to v.

Recall that our target is to compute $(1+\varepsilon)$-approximate shortest paths in $O(n \cdot \varepsilon^{-d} + n \log n)$ time. The algorithm which performs this task is given in Figure 4, in procedure SP. We separate the solution into two different cases. The first one is when $|E^{[a+1,b]}| \leq n$. In this case we can compute the shortest path from u to v in the graph $H' = (V, F^{[1,b]} \cup E^{[a+1,b]})$. Clearly, the running time is $O(n \cdot \varepsilon^{-d} + n \log n)$ and every edge from the set $E^{[a+1,b]}$ is included in the graph so there is no need to include edges from the set $F^{[b+1,c]}$.

The second, and more subtle case, is when $|E^{[a+1,b]}| > n$. In this case, edges from the set $F^{[b+1,c]}$ must be used. In order to distinguish between those edges of $F^{[b+1,c]}$ which are required and those which are forbidden, we do the following preprocessing stage. We partition the distances into at most N/n sets of size n according to their order. For the i-th set we compute all the pairs of shortest paths in the graph $H^i = (V, E^{[(i-1)n,i\cdot n]} \cup F^{[1,i\cdot n]})$ (see Figure 3(b)). We then update the distance information in the matrix M_i. For every pair $u, v \in V$ we set $M_i[u,v]$ with $min(M_{i-1}[u,v], \delta_{H^i}(u,v))$. This process is done only once before performing the actual search. It appears in Figure 4 in procedure $apBLSP$.

When searching for a specific L-bounded leg shortest path we classify the edges as follows. Let a,b and c be the same indices as before (See Figure 3(a)). Let b' be the largest multiple of n which is smaller than or equal to b. We initialize the set of edges F' with $F^{[1,b]} \cup E^{[b'+1,b]}$. A spanner edge $(p,q) \in F^{[b+1,c]}$ is added to F' if and only if $M_{b'}[p,q] \leq (1+\varepsilon)w_b$. Note that, although we add edges to F', the total number of edges is still $O(|F|)$. We now claim the main lemma of this section:

Lemma 5 *Let $u, v \in V$ and let L be a real number. Let a, b and c be indices which satisfy $w_a \leq \frac{L}{1+\varepsilon} < w_{a+1}$, $w_b \leq L < w_{b+1}$ and $w_c \leq L(1+\varepsilon) < w_{c+1}$. Let $H' = (V, F')$, where $F' = \{(p,q) \mid ((p,q) \in F^{[1,b]} \cup E^{[b'+1,b]}) \vee ((p,q) \in F^{[b+1,c]} \wedge M_{b'}[p,q] \leq (1+\varepsilon)w_b)\}$. Then, $\delta_{G^b}(u,v)/(1+\varepsilon) \leq \delta_{H'}(u,v) \leq (1+\varepsilon)\delta_{G^b}(u,v)$.*

Proof: By Corollary 3, the edges of $E^{[1,a]}$ are properly approximated by the edges of $F^{[1,b]}$. The set $E^{[b'+1,b]}$ is added to F' without effecting its size (there are at most n edges in $E^{[b'+1,b]}$). Thus, it is only left to show how to approximate the edges in $E^{[a+1,b']}$ (see Figure 3(b)). Let $(x,y) \in E^{[a+1,b']}$ and let $w(x,y)$ be its weight. Let $x = x_1, x_2, \cdots, x_\ell = y$ be the path in $H^{[1,c]}$ which approximates the distance between x and y (see Figure 2(b)). Let (x_t, x_{t+1}) be the heaviest edge on this path, and assume that

its weight is more than w_b (otherwise the whole path is in $H^{[1,b]}$). This kind of edges is the kind we would like to have in the graph H', on which we compute an approximated shortest path, although their weight is more than w_b. It follows from Corollary 4 that every other edge on the path from x to y in $H^{[1,c]}$ is of weight at most w_a. We now consider the path $x_t, x_{t-1}, \cdots, x, y, \cdots, x_{t+2}, x_{t+1}$ from x_t to x_{t+1}. Every edge on this path besides the edge (x, y) is a spanner edge which belongs to the set $F^{[1,b]}$. Let $(x, y) \in E^{[(i-1)n, i\cdot n]}$. By the choice of b', it follows that $i \cdot n \leq b'$. When computing $M_i[x_t, x_{t+1}]$ using the graph $H^i = (V, E^{[(i-1)n,i\cdot n]} \cup F^{[1,i\cdot n]})$, all the edges on the path $x_t, x_{t-1}, \cdots, x, y, \cdots, x_{t+2}, x_{t+1}$ are part of H^i. Thus, the distance between x_t and x_{t+1}, which we find and update in $M_i[x_t, x_{t+1}]$, is bounded by the length of this path. Our next target is to bound $\delta_{H^i}(x_t, x_{t+1})$. Note that the path $x = x_1, x_2, \cdots, x_\ell = y$ is of length at most $(1+\varepsilon)w(x, y)$ and the edge (x_t, x_{t+1}) is its heaviest edge. The only change between this path and the path $x_t, x_{t-1}, \cdots, x, y, \cdots, x_{t+2}, x_{t+1}$ is that instead of the edge (x_t, x_{t+1}) the path from x_t to x_{t+1} contains the edge (x, y) whose weight $w(x, y)$ is less than $w(x_t, x_{t+1})$ (see Figure 2(b)). Thus, $\delta_{H^i}(x_t, x_{t+1})$ is at most $(1+\varepsilon)w(x, y)$. Therefore, the spanner edge (x_t, x_{t+1}) will be added to F'. This completes the first part of the proof since all the edges needed from the spanner by Lemma 2 to get a $(1+\varepsilon)$-approximation for $\delta_{G^b}(u, v)$ are added to F'. We now need to bound the approximation from below. We show that edges that were added from the range $F^{[b+1,c]}$ do not create paths which are shorter than $\delta_{G^b}(u, v)/(1+\varepsilon)$. Let $(p, q) \in F^{[b+1,c]}$ be a spanner edge that was added to F'. Suppose that its weight is $w_b + \eta$, for some small $\eta > 0$. There is a path from p to q in the graph G^b of length at most $(1+\varepsilon)w_b$. This implies that any edge, that we may use while computing a shortest path in H', can reduce the length of any path by a factor of at most $1+\varepsilon$. Thus, the shortest path from u to v in $H' = (V, F')$ is of length at least $\delta_{G^b}(u, v)/(1+\varepsilon)$. □

In Figure 4 we present our algorithm. The main procedure of the algorithm is *apBLSP*. We start by computing M_i for every $1 \leq i \leq n$. The running time of this stage is $O(n^3(\log n + \varepsilon^{-d}))$. We then proceed by doing $O(n^2 \lceil \log_{1+\varepsilon} n \rceil)$ binary searches, each at a cost of $O(n \log n(\log n + \varepsilon^{-d}))$. The *Search* procedure is identical to the one given in Figure 1. The change is in the algorithm used to compute the shortest path. Instead of computing the exact shortest path in $O(n^2)$ time, we use the algorithm SP given also in Figure 4, whose running time is $O(n(\log n + \varepsilon^{-d}))$.

Note that in order to obtain a $(1+\varepsilon)$-approximation we have to perform our inner computations in a slightly more accurate fashion. When searching for the value $\delta_G(u, v)(1+\varepsilon)^j$ we compute approximated paths. Thus, we can only conclude that $\delta_{G^{I^j(u,v)-1}}(u, v) > \delta_G(u, v)(1+\varepsilon)^{j-1}$ and cannot conclude that $\delta_{G^{I^j(u,v)-1}}(u, v) > \delta_G(u, v)(1+\varepsilon)^j$ as before. Recall that, for every $i \in \mathcal{I}^j(u, v)$, we use the distance from u to v in the graph $G^{I^j(u,v)}$ to produce an approximation for $\delta_{G^i}(u, v)$. However, we can bound $\delta_{G^i}(u, v)$ from below only with $\delta_G(u, v)(1+\varepsilon)^{j-2}$. Thus the stretch is $(1+\varepsilon)^2$ instead of $(1+\varepsilon)$. To overcome this difficulty, we run the algorithm with $\varepsilon/3$. Another accuracy issue is the following: in order to ensure that every approximated path is not smaller than the shortest path it approximates, we multiply every distance found in the graph H' with $1+\varepsilon$. As a result, the stretch factor that we get is $(1+\varepsilon)^2 \leq 1+3\varepsilon$. For the above reasons, we compute a $(1+\varepsilon/9)$-spanner.

The following theorem is derived from the above lemmas and discussion:

Theorem 6 *Let V denote a set of points in a d-dimensional l_p-metric space. A data structure which approximates any bounded leg shortest path with a stretch of $(1+\varepsilon)$ can be computed in $O(n^3(\log^3 n + \log^2 n \cdot \varepsilon^{-d}))$ time. The size of the data structure is $O(n^2 \log n)$. Given $u, v \in V$ and a real number L a distance query is answered in $O(\log \log n)$ time.*

3 Near-linear algorithm for s-t bounded leg shortest path

Let V be a set of points in a d-dimensional space. Let $s, t \in V$ and let L be a real number. In the s-t Bounded Leg Shortest Path (stBLSP) problem we are given two points s and t and a real number L and we are required to produce an L-bounded leg shortest path from s to t.

In this section we present efficient algorithms for the exact stBLSP problem in the plane for norms $l_\infty(l_1)$ and l_2. The main target in such problems is to obtain a sub-quadratic running time. In particular, for l_∞ and l_1 metrics we obtain an algorithm with a running time of $O(n \log^7 n)$ and for l_2 metric we obtain an algorithm with a running time of $O(n^{4/3+\varepsilon})$, for arbitrary $\varepsilon > 0$.

```
algorithm Dijkstra(G, s, t)

d[s] = 0, d[v] = ∞  ∀v ≠ s,    S = {s}, T = P \ {s}
while t ∉ S do
        choose a pair (u, v) ∈ S × T that minimizes d[u] + w(u, v)
        d[v] ← d[u] + w(u, v)
        S ← S ∪ {v},   T ← T \ {v}
```

Figure 5: Dijkstra's algorithm.

Both algorithms are based on a clever observation made by Chan and Efrat in [8]. They pointed out that the problem of computing a shortest path from s to t can be reduced to the problem of dynamic maintenance of bichromatic closest-pair, see Figure 5 above. In the bichromatic closest-pair problem we are given two sets of points S and T and we are required to compute the closest pair of points (u, v) such that $u \in S$ and $v \in T$. Chan and Efrat showed also that using a general technique of Eppstein [12] the problem can be further reduced into the problem of designing two dynamic nearest neighbors data structures. Let $\delta_G(p, q) = d[p] + w(p, q)$, where $d[p]$ is some weight of the point p. The exact computation of $d[p]$ is given in Figure 5. The first data structure is used to find, given a point p, a point $q \in T$ minimizing $\delta_G(p, q)$. The second data structure is used to find, given a point q, a point $p \in S$ minimizing the function $\delta_G(p, q)$). If both data structures support queries, insertions and deletions in amortized time of $O(T(n))$ then the dynamic bichromatic closest-pair problem can be solved with $O(T(n) \log n)$ amortized insertion time and $O(T(n) \log^2 n)$ amortized deletion time. In order to apply this reduction to our problem we use the following distance function:

$$\delta_G(p, q, L) = \begin{cases} \delta_G(p, q) & w(p, q) \leq L \\ \infty & w(p, q) > L \end{cases}$$

This function captures also the bound on the leg. It stems from the function definition that if the closest pair is at distance of ∞ then we can stop the algorithm since no L-bounded leg path exists between s and t. We only left to show how to create two data structures as required above which work with respect to the distance function $\delta_G(u, v, L)$.

3.1 Fast shortest weighted rectilinear path: l_∞

We start by describing the first data structure in which we are given a point from S and we need to find the closest point to it from T. Note that in this case, as a result of the fact that $p \in S$ is fixed, the function $\delta_G(p, q, L)$ degenerates to $w(p, q)$. This slightly simplifies the data structure. We then proceed to the second data structure in which the function is in its general form and more complicated techniques are needed.

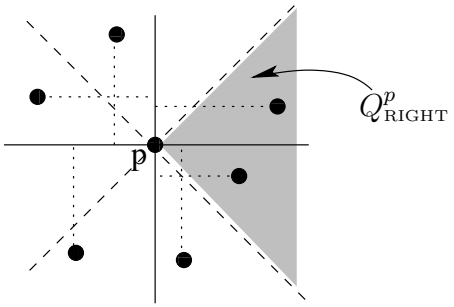

Figure 6: Division into four wedges. The triangle Q^p_{RIGHT} is shaded.

Let $p \in S$ be the query point. Let l_y and l_x be lines passing through p whose slopes are $45°$ and $135°$, respectively. These lines define four wedges which contain the triangles: $Q^p_{\text{TOP}}, Q^p_{\text{BOTTOM}}, Q^p_{\text{LEFT}}, Q^p_{\text{RIGHT}}$ (see Figure 6). The l_∞-distance from any point q which lies in $Q^p_{\text{LEFT}} \cup Q^p_{\text{RIGHT}}$ ($Q^p_{\text{BOTTOM}} \cup Q^p_{\text{TOP}}$) to p is defined by the x-distance (y-distance, respectively) to p. We will use orthogonal range tree [11] to maintain the points of T. We construct the tree in the coordinate system defined by the lines l_x and l_y. The data structure is composed from two levels. The main level is a binary search tree build on the l_x coordinate. Any node in this tree contains a binary search tree build on the l_y coordinate for all points which belong to its subtree. We refer to this tree as the secondary tree. In order to obtain the closest point we keep with every node in the secondary tree two points from its subtree, the point with the minimal x-coordinate and the point with the minimal y-coordinate in the original coordinate system. It is well known fact, see [2], that such a data structure can be build in $O(n \log n)$ time, using $O(n \log n)$ space. A range query reports all the points in a given rectangular in $O(\log^2 n + k)$ time, where k is the number of reported points. We will use the range tree to preform four queries one for each wedge.

Consider, for example, the triangle Q^p_{RIGHT}. In order to perform a query that corresponds to Q^p_{RIGHT}, we ignore the side of Q^p_{RIGHT} that is parallel to y-axis and determine all the points lying in the wedge containing Q^p_{RIGHT} by selecting $O(\log^2 n)$ nodes with disjoint canonical subsets of points located in their subtrees. Among the $O(\log^2 n)$ nodes we find the node which contains in its subtree the point with the minimal x-coordinate. This point is the closest neighbor of p in Q^p_{RIGHT}. We need to check, additionally, whether the difference between the minimum selected x-coordinate and $x(p)$ does not exceed L. If it does, we discard the selected minimum from being a candidate for a closest neighbor of p. We proceed similarly with the other triangles and their corresponding wedges. At the end, we pick the closest point to p among at most 4 candidates. We want to point out that using fractional cascading we can slightly improve the query and the update time as was done in [15]. We now proceed to the second data structure which is an extension of the first one.

Similarly to the first data structure we build an orthogonal range tree data structure for points in S in the coordinate system defined by the lines l_x, l_y. However, the previous approach will not work. Recall that in the first data structure the query point was from S, thus, its weight was not needed in order to find the closest neighbor. When the given query point is from T and we need to find the closest point from S we must consider the weight of the points of S. In other words we need to find a way to express the function $\delta_G(u, v, L)$ defined earlier.

As we already mentioned, if q is a query point its l_∞-distance to a point in Q^q_{RIGHT} is defined by the x-distance. In order to express the function $\delta_G(u, v, L)$, we make the following observation for a given query triangle Q^q_{RIGHT} (the same holds for the rest of triangles). We consider the set of functions $F = \{f_i(q) = \delta_G(p_i, q, L) \mid p_i \in Q^q_{\text{RIGHT}}\}$ and observe that this value is attained at the lower envelope of $4|F|$ segments at point $x(q)$. The corresponding y value (and therefore the closest neighbor of q in Q^q_{RIGHT}) can be found by a simple binary search over the vertices of the lower envelope.

When we query our orthogonal range tree data structure with the wedge containing triangle Q^q_{RIGHT}, we select $O(\log^2 n)$ nodes and their respective subtrees with points such that their l_∞-distance to a query point q is defined by the original x-distance to q. It may happen that few of them are of distance larger than L from q. Thus, we augment the secondary data structure by keeping, for each node w, a binary search tree T_w for original x-coordinates of the points corresponding to the nodes in the subtree rooted at w. (For wedges containing triangles Q^q_{TOP} and Q^q_{BOTTOM}, the tree T_w is built for original y-coordinates of points). Also, each node u in T_w will contain a data structure for lower envelope of functions $f_i(q)$ such that the original x-coordinate of point p_i is located in the subtree rooted at u.

A data structure for maintaining the lower envelope of n segments under deletions only was given by Chazelle [9] and Hershberger and Suri [14]. It has an amortized deletion time of $O(\log n)$. It can be transformed to handle insertions as well by a standard Bentley and Saxe binary-counting trick [4] (see also [6]) which leads to a slow down in additional logarithmic factor in updates time, i.e. insertions and deletions of points can be performed in $O(\log^2 n)$ amortized time.

In order to find a closest neighbor to q in Q^q_{RIGHT} we proceed as following. First we determine the $O(\log^2 n)$ nodes corresponding to points located at the wedge containing Q^q_{RIGHT}. Next, for every w belonging to the selected set of nodes we find a set U of $O(\log n)$ nodes in T_w corresponding to points at the wedge at distance at most L from a query point q. It is done by a binary search over T_w looking for a value $x(q)+L$. For every $u \in U$ we perform a binary search over the stored lower envelope of segments at u as described above. The closest neighbor of q is the one with the minimal value that we obtain.

Every insertion or deletion requires $O(\log^5 n)$ time, since we have $O(\log^3 n)$ nodes to update. Using the result of Eppstein [12], we obtain that the dynamic bichromatic closest-pair problem in our case can be solved in $O(\log^6 n)$ amortized insertion time and $O(\log^7 n)$ amortized deletion time. Putting it all together we obtain a $O(n \log^7 n)$ runtime solution for our problem, since each iteration of Dijkstra's algorithm requires $O(\log^7 n)$ amortized time. The above algorithm can be easily generalized to work in any $\mathcal{R}^d, d > 2$ with running time $O(n \log^{d+5} n)$. We notice that at the same time we can produce the actual path by keeping at each iteration the closest-pair information. We also should point out that we have not made a serious attempt to improve the performance of the above searching procedure, because our main interest was in obtaining a near-linear solution of the problem.

Remark: For l_2 metric we can use both data structures developed in [8] that are produced by the technique of Agarwal et al. [1]. The first data structure is obtained by considering the lower envelope of the bivariate functions $\{g_i(p) = w(p, q_i) | q_i \in T\} \cup \{g'(p) = L\}$. The function $g'(p)$ corresponds to restriction that $w(p, q_i) \leq L$. The second data structure is obtained by considering the lower envelope of the bivariate functions $\{f_i(q) = \delta_G(p_i, q, L) | p_i \in S\} \cup \{f'_i(q) = d[p_i] + L | p_i \in S\}$. The restriction $w(p_i, q) \leq L$ is expressed by functions $f'_i(q)$. Both data structures have $T(n) = O(n^{1/3+\varepsilon})$ query and update times which leads to overall $O(n^{4/3+\varepsilon})$ runtime for the entire problem, since we have $O(n)$ deletions and insertions.

We end this section with the following Theorem:

Theorem 7 *There exists an algorithm for the stBLSP problem in l_1 and l_∞ metrics that, given $s, t \in V$ and a real number L, computes in $O(n \cdot polylog(n))$ time the exact L-bounded shortest path from s to t.*

4 Bounded unit leg shortest path

In order to find the shortest unweighted path under the restriction that the longest edge can have weight at most L we designed more efficient algorithms in terms of runtime and space. In what follows we show how to obtain either faster runtime or linear space solutions. The presented solutions are for planar case and can be easily generalized to deal with d-dimensional point sets, $d > 2$.

4.1 Bounded unit leg shortest rectilinear path

We apply a BFS-based scheme in order to determine the required path. We start with point s and look for all points in S that are of distance at most L from s. In order to find these points we draw a square Q of side length $2L$, centered at s and ask a question "Which points are inside of Q?". After locating all the relevant points we put all of them into the BFS queue and continue the same process until we meet t. The problem with this approach is that a large number of points can reappear in the queue many times, thus, leading to quadratic time algorithm. To avoid this problem, we will again make a use of standard orthogonal range trees for two dimensional range searching. However, we do not apply the fractional cascading technique for speed up the algorithm, since it does not allow to perform a marking process efficiently as described below.

The idea is to mark relevant nodes (in secondary data structure of orthogonal range tree) when querying and reporting points, that will prevent from points to reappear over and over in the BFS queue. Recall that the orthogonal range tree is composed from two levels. The main level is a binary search tree build on the x coordinate. Any node in this tree contains a binary search tree (secondary data structure) build on the y coordinate for all points which belong to its subtree. Notice that the number of secondary data structure tree is $O(n)$. For a current query, square Q, first we compute the canonical subsets the represent our output set of points. Overall we might have $O(\log^2 n)$ such subsets each one representing a subtree of some secondary data structure. Next, for each node u that is a root of some canonical set U_c, we check whether u or some of its ancestors in the tree it belongs to is marked. If yes, we can throw out the entire subtree U_c since we know that the points belonging to this set already have been appeared in the BFS queue. Otherwise, we mark u as visited and insert all the points from U_c to the BFS queue. Notice, that since the depth of the primary tree and each one of the secondary data structure trees is at most $O(\log n)$, every point can reappear in the BFS queue at most $O(\log^2 n)$ times. The additional time to be spend per each of $O(\log^2 n)$ roots of canonical subtrees in the secondary data structures is $O(\log n)$ when checking up ancestors. Thus, we conclude that the total runtime of the proposed scheme is $O(n \log^5 n)$.

4.2 Linear space shortest Euclidean path

The basic algorithm remains the same; the query, however, transforms from square with side length $2L$ to disk of radius L. We use the following useful observation that is crucial in our algorithm. The range searching with balls in $\mathcal{R}^d$ can be formulated as an instance of halfspace range searching in $\mathcal{R}^{d+1}$. The technique is called *linearization* and in general, balls in $\mathcal{R}^d$ admit a linearization of dimension $d+1$, see [2]. The halfspace range searching problem in $\mathcal{R}^d$ is defined as follows. Given a set of n points in d-dimensional space, build a data structure that allows us to determine efficiently how many points lie in a query halfspace. We will use a data structure, called *partition tree*, being proposed by Matoušek [16] that is based on simplicial partition for a set of points, can be build in $O(n \log n)$ time and $O(n)$ space and answers queries in $O(n^{1-1/d+\varepsilon}+k)$ time. We notice, that there are more efficient data structures with faster optimal $O(\log n + k)$ query time, see [10, 3]; however the output produced by these data structures can not be represented as a collection of disjoint canonical subsets which is a main ingredient of our algorithm. In contrary, in our case, for a query halfspace h in $\mathcal{R}^3$, a partition tree allows to select a set W of $O(n^{2/3})$ nodes from the tree with the property that the subset of points in h is the disjoint union of the canonical subsets of the selected nodes. The basic idea in construction of partition tree for a given set of n points is to find a division of the plane into several regions, each of which contains at least a constant fraction of the points, and such that any line misses one (or several) of the regions. Given a query halfplane, we can then treat the points in the regions missed by its boundary line very efficiently: They either all lie inside our halfplane, or all outside. Thus it remains to handle the points in the regions intersected by the boundary line of the query halfplane. To this end, the partition scheme is applied recursively for subsets in each of the regions, until we reach trivially small subsets (of a constant size). Matoušek [16] has

shown that it is possible to construct a partition tree of depth $O(\log\log n)$ such that query selects at most $O(n^{2/3})$ disjoint canonical subtrees of the partition tree. Therefore, as in previous case, we can perform a BFS-like algorithm with marking nodes by taking into account that the depth of partition tree is at most $O(\log\log n)$. The overall running time stands at $O(n^{5/3+\varepsilon})$.

Remark: All algorithms described in this section can be easily transformed to deal with a lower bound constraint on the edges of the required path, i.e. the path should contain edges of weight at least L'.

5 Space efficient data structure for answering path queries

We now turn to show how using only $\tilde{O}(n^{2.5})$ space instead of $\tilde{O}(n^3)$ it is still possible to produce a path of stretch $(1+\varepsilon)$. This result applies also to weighted directed graphs. However, the running time for this special construction is $\tilde{O}(n^4)$. Let $I^j(u,v) \in \mathcal{I}(u,v)$. We augment the information that we already save with $I^j(u,v)$ with the following: If $h_{G^{I^j(u,v)}}(u,v) \leq \sqrt{n}$ we simply keep the whole path from u to v in $G^{I^j(u,v)}$. If $h_{G^{I^j(u,v)}}(u,v) > \sqrt{n}$ then we slice the path into intervals of $\sqrt{n}$ vertices and save only those vertices that separate two consecutive intervals. Additionally, in this case, we take the minimal index $i \in \mathcal{I}^j(u,v)$ such that $h_{G^i}(u,v) \leq \sqrt{n}$ and save the shortest path from u to v in the graph G^i. Note that such an index does not necessarily exists.

We now describe how to produce a path of stretch $(1+\varepsilon)$ using the data structure described above.

Let $u, v \in V$ and let L be a real number. Let i be the index that satisfies $w_i \leq L < w_{i+1}$. Let $I^j(u,v)$ be the index of the graph used to approximate the L-bounded leg shortest path from u to v in the graph G^i. If $h_{G^{I^j(u,v)}}(u,v) \leq \sqrt{n}$ then we simply output the path that we have saved with $I^j(u,v)$. The more subtle case is when $h_{G^{I^j(u,v)}}(u,v) > \sqrt{n}$. Let $u = x_1, x_2, \cdots, x_\ell = v$ be the vertices saved with $I^j(u,v)$. By definition we know that $h_{G^{I^j(u,v)}}(x_k, x_{k+1}) \leq \sqrt{n}$, for every $k \in [1,l)$. Let $I^{j_k}(x_k, x_{k+1})$ be the largest index in $\mathcal{I}(x_k, x_{k+1})$ which is smaller than $I^j(u,v)$. In other words, we find the set $\mathcal{I}^{j_k}(x_k, x_{k+1})$ which contains $I^j(u,v)$. By Lemma 1 we get $\delta_{G^{I^{j_k}(x_k,x_{k+1})}}(x_k, x_{k+1}) \leq (1+\varepsilon)\delta_{G^{I^j(u,v)}}(x_k, x_{k+1})$. Summing it up we get:

$$\sum_{k=1}^{\ell-1} \delta_{G^{I^{j_k}(x_k,x_{k+1})}}(x_k, x_{k+1}) \leq (1+\varepsilon) \sum_{k=1}^{\ell-1} \delta_{G^{I^j(u,v)}}(x_k, x_{k+1}) = (1+\varepsilon)\delta_{G^{I^j(u,v)}}(u,v)$$

Thus, we get that using the vertices $x_1, x_2, \cdots, x_\ell$ we can obtain a path of length $(1+\varepsilon)\delta_{G^{I^j(u,v)}}(u,v)$. And in total we have a path of length $(1+\varepsilon)^2\delta_{G^i}(u,v)$. However, we are not done yet. Two issues need to be addressed. The first issue is how to obtain the index $I^{j_k}(x_k, x_{k+1})$ for every $k \in [1,l)$. This is done in a similar manner to the query procedure. Given the index $I^j(u,v)$ we do a binary search among the values of $\mathcal{I}(x_k, x_{k+1})$ at a cost of $O(\log\log n)$. The second issue is which path we actually output for every pair of vertices x_k and x_{k+1}. It is possible that the path from x_k to x_{k+1} in $G^{I^{j_k}(x_k,x_{k+1})}$ contains more than $\sqrt{n}$ edges. But recall that in such a case we save with $I^{j_k}(x_k, x_{k+1})$ a path from the graph G_r where r is the minimal index in $\mathcal{I}^{j_k}(x_k, x_{k+1})$ which satisfies $h_{G^r}(u,v) \leq \sqrt{n}$. We know that such an index must exists since $I^j(u,v) \in \mathcal{I}^{j_k}(x_k, x_{k+1})$ and $h^{I^j(u,v)}(x_k, x_{k+1}) \leq \sqrt{n}$.

Acknowledgments. The authors like to thank Haim Kaplan for introducing between them. The first author like to thank to Uri Zwick, Vera Asudi and Eyal Lubetzky for their helpful remarks.

References

[1] Pankaj K. Agarwal, Alon Efrat, and Micha Sharir. Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications. *SIAM Journal on Computing*, 29:912–953, 1999.

[2] Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, Providence, RI, 1999.

[3] A. Aggarwal, M. Hansen, and T. Leighton. Solving query-retrieval problems by compacting Voronoi diagrams. In *Proc. 22nd Annu. ACM Sympos. Theory Comput.*, pages 331–340, 1990.

[4] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformations. *J. Algorithms*, 1:301–358, 1980.

[5] P. Bose, A. Meheswari, G. Narasimhan, M. Smid, and N. Zeh. Approximating geometric bottleneck shortest paths. *Computational Geometry: Theory and Applications*, 29:233–249, 2004.

[6] T. M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. *J. ACM*, 48:1–12, 2001.

[7] Timothy M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. *ACM-SIAM Symposium on Discrete Algorithms*, 2006.

[8] Timothy M. Chan and Alon Efrat. Fly cheaply: On the minimum fuel consumption problem. *J. Algorithms*, 41:330–337, 2001.

[9] Bernard Chazelle. On the convex layers of a planar set. *IEEE Trans. Inform. Theory*, IT-31(4):509–517, July 1985.

[10] Bernard Chazelle and F. P. Preparata. Halfspace range search: An algorithmic application of k-sets. *Discrete Comput. Geom.*, 1:83–93, 1986.

[11] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

[12] D. Eppstein. Dynamic Euclidean minimum spanning trees and extrema of binary functions. *Discrete Comput. Geom.*, 13:111–122, 1995.

[13] D. Eppstein. Spanning trees and spanners. In Handbook of Computational Geometry, J.-R. Sack, and J. Urrutia, editors, *Elsevier*. 2000.

[14] J. Hershberger and S. Suri. Applications of a semi-dynamic convex hull algorithm. *BIT*, 32:249–267, 1992.

[15] S. Kapoor and M. Smid. New techniques for exact and approximate dynamic closest-point problems. *SIAM J. Comput.*, 25:775–796, 1996.

[16] J. Matoušek. Efficient partition trees. *Discrete Comput. Geom.*, 8:315–334, 1992.

[17] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1):47–74, 2004.

[18] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999.

[19] P. M. Vaidya. A sparse graph almost as good as the complete graph on points in K dimensions. *Discrete & Computational Geometry*, 6:369–381, 1991.

[20] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, 2002.