Fast Approximate Lipschitz Extensions in Doubling Metrics

Donald R. Sheehy*

Abstract

We give an efficient algorithm for computing Lipschitz extensions in doubling metrics. After $O(n \log n)$ preprocessing time for sets A and B of size n, any t-Lipschitz function on A can be extended to B in $O(n)$ time. The extension is approximate in the sense that it may be at most $(1+\varepsilon)t$-Lipschitz.

1 Introduction

A real-valued function on a metric space is t-Lipschitz if for all points x and y, we have $f(x) \leq f(y) + t\,\mathrm{d}(x, y)$. Let A and B be sets of points in a metric space. The *Lipschitz Extension Problem* is to extend a t-Lipschitz function $f : A \to \mathbb{R}$ to a function $f^+ : A \cup B \to \mathbb{R}$ that is also t-Lipschitz and agrees with f at points of A. In this paper, we give efficient algorithms for computing approximate Lipschitz extensions in metric spaces that exhibit intrinsic low-dimensionality in the form of bounded doubling dimension. This is a generalization of low-dimensional Euclidean spaces that allows for packing arguments similar to geometric volume-packing arguments.

More specifically, we will construct the maximum and minimum Lipschitz extensions. The main application of this work is in topological data analysis where it was shown that topological invariants called *subbarcodes* of an unknown function can be computed from upper and lower bounds on that function [2]. Thus, for a Lipschitz function known only at a subset of points, the algorithms efficiently compute upper and lower bounds on the function values at other points.

The paper provides two specific contributions to the algorithmic theory of Lipschitz extensions. The first is a direct reduction from the problem of computing maximum Lipschitz extensions to the all-nearest neighbor problem. This involves constructing a new metric from the input points and the function f so that the maximum Lipschitz extension can be computed from the nearest neighbors in the new metric. The second contribution is an approximation algorithm that can compute Lipschitz extensions in linear time for any function after $O(n \log n)$ preprocessing time. That is, we pay $O(n \log n)$ time once and then can compute the maximum (or minimum) Lipschitz extension of any function in linear time. This requires a new notion of approximation where the output is guaranteed to lie between the maximum t-Lipschitz extension and the maximum $(1+\varepsilon)t$-Lipschitz extension.

The approach is based on dual-tree algorithms on greedy trees. The background on doubling metric spaces, Lipschitz extensions, and greedy trees is given in Section 2. We then detail the relationship between maximum Lipschitz extension and nearest neighbor search in Section 3. Finally, in Section 4, we explain how dual-tree algorithms can be used to compute the Lipschitz extensions exactly and approximately with good bounds on the running time.

Part of the novelty in this approach comes from the fact that the usual methods to reduce the search space in dual-tree algorithms don't apply directly for Lipschitz extensions. In standard dual-tree applications like all-nearest neighbor search, one can prune neighbors that are far away. However, the maximum Lipschitz extension of a point can be determined be a far away point even if there are other points nearby.

2 Background

2.1 Metric Spaces and Doubling Dimension

A metric space is a pair (X, d) where X is a set and $\mathrm{d} : X \times X \to \mathbb{R}$ is the *metric* or *distance function*. We denote the distance from $x \in X$ to any $S \subseteq X$ as $\mathrm{d}(x, S) = \min_{s \in S} \mathrm{d}(x, s)$.

The *diameter* of a set $A \subseteq X$ is the maximum pairwise distance of points in the set and is denoted diameter(A). A collection of subsets of A is called a *cover* if the union is all of A. The *diameter of a cover* is the maximum diameter among the sets in the cover. A *δ-cover* is a cover of diameter at most δ. For a set $A \subseteq X$ and $\delta \in \mathbb{R}$, the *δ-covering number* is the minimum number of sets in any δ-cover.

The *doubling constant* for a metric space is the maximum over all subsets $A \subseteq X$ of the (diameter$(A)/2$)-covering number. The *doubling dimension* is the base-2 logarithm of the doubling constant.

Note that in many works, the doubling dimension is defined in terms of the radius of a ball rather than the diameter of an arbitrary set. There are some conveniences to that approach, but it leads to certain difficulties when reasoning about the doubling dimension of a subset. Specifically, if the doubling dimension is defined

*Computer Science Department, North Carolina State University don.r.sheehy@gmail.com

in terms of radii instead of diameters, then the dimension is not monotone with respect to subsets—removing points can increase the dimension. The definition given here is the original one [6] and should be preferred.

The following lemma is the standard packing argument that motivates the use of doubling dimension in many settings. It is the main tool for bounding the size of sets in doubling metrics.

Lemma 1 (Standard Packing Lemma) *Let P be a finite metric space of doubling dimension d. If D is the diameter of P and for all distinct $a, b \in P$, we have $\mathrm{d}(a,b) > \delta$, then*

$$|P| \leq \left(\frac{2D}{\delta}\right)^d.$$

Proof. Here is the standard proof included for completeness. If $\rho = 2^d$ is the doubling constant, then P can be covered by ρ sets of diameter $D/2$. Covering these sets gives a $D/4$ cover of size ρ^2. Repeating this process until the diameters are less than δ results in cover where each point of P is in its own set. This requires $k = \lceil \lg D/\delta \rceil$ iterations and gives a total size

$$|P| \leq \rho^k = 2^{dk} \leq 2^{d\lceil \lg D/\delta \rceil} \leq \left(\frac{2D}{\delta}\right)^d.$$

□

2.2 Lipschitz Functions and their Extensions

For any set X, there is a natural partial order on real valued functions $f, g : X \to \mathbb{R}$ defined as

$$f \leq g \text{ if and only if for all } x \in X,\ f(x) \leq g(x).$$

Definition 2 *A function $f : X \to \mathbb{R}$ is* t-Lipschitz *iff for all $a, b \in X$,*

$$|f(a) - f(b)| \leq t\mathrm{d}_X(a,b).$$

Equivalently, for all $a, b \in X$,

$$f(b) - t\mathrm{d}_X(a,b) \leq f(a) \leq f(b) + t\mathrm{d}_X(a,b).$$

A function is Lipschitz *if it is 1-Lipschitz.*

Let $A \subseteq X$ be any subset of a metric space and let $f : A \to \mathbb{R}$ be any function. An extension of f to X is a function $f^+ : X \to \mathbb{R}$ such that for all $a \in A$, we have $f^+(a) = f(a)$. If f is t-Lipschitz, then there always exists a t-Lipschitz extension. Perhaps the simplest proof of this statement is to construct such an extension. In fact, we will construct two such extensions with certain universal properties as described in the following lemma.

Lemma 3 *For any metric space X and any t-Lipschitz function $f : A \to X$ with compact*[1] *$A \subseteq X$, there exist t-Lipschitz extensions $\hat{f}, \check{f} : X \to \mathbb{R}$ such that for all t-Lipschitz extensions g, we have $\hat{f} \leq g \leq \check{f}$.*

Proof. Define the *minimum Lipschitz extension* as

$$\hat{f}(x) = \max_{a \in A} f(a) - t\mathrm{d}(x,a).$$

Define the *maximum Lipschitz extension* as

$$\check{f}(x) = \min_{a \in A} f(a) + t\mathrm{d}(x,a).$$

Let $g : X \to \mathbb{R}$ be any t-Lipschitz extension of f. By the definition of Lipschitz,

$$f(a) - t\mathrm{d}(x,a) \leq g(x) \leq f(a) + t\mathrm{d}(x,a)$$

for all $x \in X$ and all $a \in A$. So,

$$g(x) \geq \max_{a \in A} f(a) - t\mathrm{d}(x,a) = \hat{f}(x),$$

and

$$g(x) \leq \min_{a \in A} f(a) + t\mathrm{d}(x,a) = \check{f}(x).$$

□

2.3 Greedy Permutations and Greedy Trees

Given any ordered set of points $P = \{p_0, \ldots, p_{n-1}\}$ in a metric space, we define the ith *prefix* to be

$$P_i := \{p_0, \ldots, p_{n-1}\}.$$

We say that the permutation P is *greedy* if for all $i > 0$, we have

$$\mathrm{d}(p_i, P_i) = \max_{q \in P} \mathrm{d}(q, P_i).$$

In other words, every point is the farthest point to its prefix. Also known as *Gonzalez ordering* or *farthest point traversals*, the greedy permutation is a standard way to produce samples that satisfy both packing and covering properties.

The permutation P is *β-approximate greedy* if for all $i > 0$, we have

$$\beta\mathrm{d}(p_i, P_i) \geq \max_{j \geq i} \mathrm{d}(p_j, P_i).$$

As the definition of (approximate) greedy permutations depends on the distance of each point to its nearest predecessor, it is natural to associate each point with an approximate nearest predecessor called its *parent*. The *insertion distance* of p_i is defined as

$$\varepsilon_i := \mathrm{d}(p_i, \mathrm{par}(p_i)).$$

[1]Here we are assuming A is compact. This is justified because for all of our applications, A will be finite. If one wants to consider these definitions for extensions of noncompact sets, then one simply replaces the max and min with sup and inf.

By convention, we set $\varepsilon_0 = \infty$. For β-approximate greedy permutations we require that parents are β-approximate nearest predecessors, i.e., for all $i > 0$,

$$\varepsilon_i \leq \beta \mathrm{d}(p_i, P_i).$$

Moreover, we require that the insertion distances are in non-increasing order, i.e., that $i < j$ implies $\varepsilon_i \geq \varepsilon_j$. The parent relation turns P into a tree called the *greedy tree.*

Approximate greedy permutations satisfy the following packing bounds.

Lemma 4 *Let P be a β-approximate greedy permutation with insertion distances ε_i. For all i, j, k such that $0 \leq i < j \leq k$, we have*

$$\mathrm{d}(p_i, p_j) \geq \varepsilon_k / \beta.$$

Proof. We use the ordering on insertion distances, the approximate nearest neighbor property of parents, and the fact that $p_i \in P_j$ to derive the bound as follows.

$$\varepsilon_k \leq \varepsilon_j \leq \beta \mathrm{d}(p_j, P_j) \leq \beta \mathrm{d}(p_j, p_i).$$

□

We say that the greedy tree is *α-scaling* iff for all a, b, c such that $a = \mathrm{par}(b)$ and $b = \mathrm{par}(c)$, we have $\mathrm{d}(b, c) \leq \alpha \mathrm{d}(a, b)$. That is, the distance between points is decreasing by a factor of α in each level of the tree.

Greedy permutations and greedy trees can be constructed in $O(n \log n)$ time in doubling metrics [1, 5].

3 Lipschitz Extension as Nearest Neighbor Search

In this secction, we show how the Lipschitz extension problem can be described exactly as an all-nearest-neighbors problem in a new metric space defined by the input function f.

Let A and B be finite subsets of a metric space (X, d). Let $f : A \to \mathbb{R}$ be t-Lipschitz. For any point $b \in B$, computing the maximum t-Lipschitz extension means computing

$$\check{f}(b) = \min_{a \in A} f(a) + t\mathrm{d}(a, b).$$

There is a direct reduction from this problem to nearest neighbor search. Recall that a nearest neighbor of b in A is defined as a point a minimizing $\mathrm{d}(a, b)$. There is an extensive literature on this problem. In this case, we want to minimize a kind of weighted distance where the value $f(a)$ influences the distance.

Define the new metric

$$\mathrm{d}'(x, y) := t\mathrm{d}(x, y) + |f(x) - f(y)|.$$

If $b \in B$ then $f(b)$ is unknown. In that case, let $w = \min_{a \in A} f(a)$ be the smallest value of f and let

$$\mathrm{d}'(a, b) := t\mathrm{d}(a, b) + |f(a) - w|.$$

Note that d' is a proper metric as it is formed from the sum of two other metrics. It can be viewed as the L_1-product metric of d (scaled by t) and the standard metric on $\mathbb{R}$. From this perspective, the points are viewed as pairs $(a, f(a))$ or (b, w).

The following proposition shows how the nearest neighbor with respect to d' determines the maximum Lipschitz extension.

Proposition 5 *For any $b \in B$, let $a \in A$ be the nearest neighbor of b with respect to the distance d'. Then, $\check{f}(b) = f(a) + t\mathrm{d}(a, b)$.*

Proof. Suppose for contradiction that there exists some $a' \in A$ such that

$$\check{f}(b) = f(a') + t\mathrm{d}(a', b) < f(a) + t\mathrm{d}(a, b).$$

It follows that

$$\mathrm{d}'(a', b) < \mathrm{d}'(a, b)$$

and thus, a was not the nearest neighbor of b with respect to d', a contradiction. □

The Trouble With (Approximate) Nearest Neighbors Although the reduction from Lipschitz extension to nearest neighbors gives correct output, it suffers three substantial drawbacks.

1. The nearest neighbor search structure must be reconstructed for each function f.

2. The dimension can increase.

3. Approximate nearest neighbors do not give an approximation to the Lipschitz extension.

This last point is the most critical. Most nearest neighbor data structures that achieve fast running times only give approximate answers. Suppose a is only a c-approximate nearest neighbor of b with respect to d'. Then, for all $a' \in A$,

$$\mathrm{d}'(a, b) \leq c\mathrm{d}'(a', b).$$

There is no direct way to translate this kind of multiplicative approximation to a corresponding multiplicative approximation for the Lipschitz extension. Indeed, if the function value is close to zero, small absolute error may represent a large relative error. This challenge motivates our new notion of approximate Lipschitz extension in the next section.

4 Dual-Tree Algorithms

This section presents algorithms for exact and approximate Lipschtiz extension. The inputs are metric spaces A and B as well as a function $f : A \to \mathbb{R}$. The input metric spaces have been preprocessed into greedy trees. The output is a Lipschtiz extension $g : B \to \mathbb{R}$.

4.1 Traversing Greedy Trees

Given an α-scaling, β-approximate greedy permutation, we process it into a list of 4-tuples of the form `(q, parent, radius, par_radius)`, where

- `q` is the next point,
- `parent` is the parent of `q`,
- `radius` is the length of the longest path from `q` to a leaf, and
- `par_radius` is the length of the longest path from parent to a leaf that only include points appearing *after* `q`.

The length of a path is caclulated as the sum of the lengths of the edges. The only element of this list that is not self-explanatory is the last one. The purpose of `par_radius` is to update the radius of the parent after after inserting `q`. Each insertion can be viewed as removing the subtree rooted at `q` and so the inserted points are all identified with trees that partition P. Thus, removing a subtree from the parent may change its radius.

The 4-tuples are ordered according to the given greedy permutation, so that for each i, we let

$$\varepsilon_i := \mathrm{d}(p_i, \mathrm{par}(p_i)),$$

and have $\varepsilon_0 \geq \varepsilon_1 \geq e_2 \geq \cdots$.

The list of 4-tuples encodes the traversal of the greedy tree. The radii are computed by iterating through the permutation in reverse order. This reversal is like building up the tree from subtrees rather than deconstructing the greedy tree into individual points as will happen in the algorithm. Each time a point is encountered, the radius of its parent is exactly the length of the longest path from the parent to a leaf using only points that come later in the order. The pseudocode is below.[2]

```
def traverse(P):
    radius = {p: 0 for p in P}
    par_radius = {}
    for q in reversed(P):
        p = P.par(q)
        par_radius[q] = radius[p]
        radius[p] = max(radius[p],
                        radius[q] + d(p, q)
                        )
    return [(p, P.par(p), radius[p],
             par_radius[p]
            )
            for p in P]
```

[2]The syntax guide for this style of pseudocode can be found at `python.org`.

By definition, the maximum Lipschitz extension at $b \in P_b$ is defined in terms of a point $a \in P_a$. We say the pair (a, b) is *critical* if

$$\check{f}(b) = f(a) + \mathrm{d}(a, b).$$

There is a trivial quadratic-time algorithm to compute the maximum Lipschitz extension. For each $b \in B$, iterate over the points of A to search for the point a such that (a, b) is critical.

In this section we give two dual-tree algorithms [3,4], one for the exact maximum 1-Lipschitz extension and one for an approximation. The general paradigm of dual-tree algorithms gains efficiency by exploiting the spatial locality inherent in performing many searches on the same tree. The greedy trees represent clusters and the tree structure gives a hierarchy of clusterings. The two trees are traversed simultaneously. In both algorithms, we speed up the naive quadratic-time algorithm by comparing entire clusters of points instead of treating them individually.

4.2 An Exact Algorithm

In this subsection, we give a high-level description of a dual-tree algorithm for the exact maximum Lipschitz extension problem. Later, we give a much more detailed description of the approximate version.

It suffices to consider only 1-Lipschitz functions as one can scale distances and function values for different Lipschitz constants. Also, we don't consider minimum extensions separatly because the min extension of f is the negation of the max extension of $-f$.

The algorithm inserts points one at a time. Each inserted point x has an associated radius $r[x]$ that goes down as the cluster at x shrinks (i.e., as more points are inserted). Let $a \in A$ and $b \in B$ be centers of nodes with radii $r[a]$ and $r[b]$ respectively. Let P_a and P_b denote the corresponding clusters, i.e., the points of the subtrees rooted at the nodes.

The main data structure used by the algorithm is the *viability graph* V, a bipartite graph on nodes maintaining the invariant that if there exists a critical pair $(a' \in P_a, b' \in P_b)$, then there is an edge $a \sim b$ in V. We also store for each point $b \in B$, the current upper bound

$$g[b] := \min_{a \sim b} f(a) + \mathrm{d}(a, b) \geq \check{f}(b).$$

Note that for $r[b]$ and $g[b]$ we use square brackets to remind the reader that these are not fixed functions, but rather dictionaries that are updated as the algorithm progresses.

The points are processed in order of insertion distance by merging the two greedy permutations. For each point b in B that has been inserted, we maintain the invariant that $g[b] \geq \check{f}(b)$. The viability graph starts with a single edge connecting the roots of the two trees. In each iteration, the new point is assigned to have the same neighbors as its parent. The radius of the new point and its parent are updated. For any point b whose neighbors changed (including if it was just inserted), we check to see if the new neighbor reduced $g[b]$. The last step is to prune away edges incident to the new nodes that are no longer needed to satisfy the viability graph invariant. We show that the condition (1) below suffices to guarantee an edge can be safely pruned.

The viablity graph invariant allows that if there are no critical pairs in $P_a \times P_b$, then $a \sim b$ can be pruned. The trick is to prune these edges without searching for a critical pair among the subtrees. Any edge $a \sim b$ satisfying the following condition can be pruned.

$$g[b] + r[b] < (f(a) - r[a]) + (\mathrm{d}(a,b) - r[a] - r[b]). \quad (1)$$

This condition is exactly what is required to guarantee the non-existence of a critical pair in $P_a \times P_b$. By the triangle inequality and the definition of Lipschitz, we can see the following bounds hold for all $a' \in P_a$ and $b' \in P_b$ whenever (1) holds.

$$\begin{aligned}\check{f}(b') &\leq g[b] + r[b] \\ &< (f(a) - r[a]) + (\mathrm{d}(a,b) - r[a] - r[b]) \\ &\leq f(a') + \mathrm{d}(a',b').\end{aligned}$$

So, (a', b') is not critical.

The correctness of the algorithm follows immediately from the viability graph invariant. At the end of the algorithm, all clusters are single points and there is an edge if and only if the the points are critical. The upper bounds give the exact maximum Lipschitz extension.

The running time is dominated in each iteration by the work of iterating over the new edges and checking if they can be pruned. Although one might expect that many edges can be pruned in practice, there is no clear way to *guarantee* that only a subquadratic number of edges will be considered in the course of the algorithm. Unlike in the all-approximate-nearest-neighbor problem, the critical pairs can be far apart. It seems unlikely that any dual-tree methods will give subquadratic running times for the *exact* Lipschitz extension problem. The next sections show that for approximations, the viability graph remains sparse.

An Approximation Algorithm The goal of the approximation algorithm is to produce a function $g : B \to \mathbb{R}$ such that $\check{f} \leq g \leq \check{f}_{1+\varepsilon}$ for a given $\varepsilon \geq 0$, where

$$\check{f}_{1+\varepsilon}(b) = \min_{a \in A} f(a) + (1+\varepsilon)\mathrm{d}(a,b).$$

The approximation algorithm is the same as the exact algorithm, except that we will modify the pruning condition. If the following condition is satisfied, then the edge $a \sim b$ can be pruned.

$$g[b]+r[b] < (f(a)-r[a])+(1+\varepsilon)(\mathrm{d}(a,b)-r[a]-r[b]). \quad (2)$$

Note that for $\varepsilon = 0$, this condition is the same as the exact case. However, this new pruning condition requires a different correctness proof and allows for a stronger running time guarantee as we will see below. This algorithm can and often will prune away even critical edges, but not before an approximate value of g has been recorded.

For completeness, we give the pseudocode below for edge pruning as a method of the viability graph. For simplicity, we assume the value of ε is fixed.

We assume a basic bipartite graph data structure that is initialized with a pair of vertex sets and supports the operations `add_edge`, `remove_edge`, and `nbrs`. The first two are self-explanatory. The function `nbrs` returns an iterator over the neighbors of a vertex.

```
def try_prune(self, a, b, r, g, f):
    if g[b] + r[b] < (f(a) - r[a])
        + (1+epsilon)(d(a,b) - r[a]- r[b]):
        self.remove_edge(a,b)

def lipschitz_extend(A, B, f):
    V = ViabilityGraph(A,B)
    V.add_edge(A[0], B[0])

    # Initialize r and g
    r = dict()
    g = dict()
    g[B[0]] = f(A[0]) + d(A[0], B[0])

    # order merged lists by insertion distance
    L = merge(traverse(A), traverse(B))

    for (q, parent, radius, par_radius) in L:
        r[parent] = par_radius
        r[q] = radius

        if parent in g:  # (if q in B)
            for x in V.nbrs(parent):
                g[q] = min(g[q], f(x) + d(x,q))
                V.add_edge(x,q)
                V.try_prune(x,q,r,g,f)
                V.try_prune(x,parent,r,g,f)
        else: # (if q in A)
            for x in V.nbrs(parent):
```

```
            g[x] = min(g[x], f(q) + d(q,x))
            V.add_edge(q,x)
            V.try_prune(q,x,r,g,f)
            V.try_prune(parent,x,r,g,f)

    return g
```

4.3 A Proof of Correctness

Theorem 6 *The output of* `lipschitz_extend(A,B,f)` *is a function $g : B \to \mathbb{R}$ such that*

$$\check{f} \le g \le \check{f}_{1+\varepsilon}.$$

Proof. At any time in the algorithm, let $A' \subseteq A$ and $B' \subseteq B$ be the points that have been inserted so far. We prove that after each iteration of the main loop, we satisfy the invariant that for all $b \in B'$, we have

$$\check{f}(b) \le g[b] \le \min_{a \in A'} (f(a) + (1+\varepsilon)\mathrm{d}(a,b)).$$

In other words, the current extension g on B' is always at least as large as the maximum Lipschitz extension and is always at most as large as the maximum $(1+\varepsilon)$-Lipschitz extension of the points that have been inserted so far. This invariant will imply the desired guarantee.

Let $a' \in A'$ and $b' \in B'$ be such that a' minimizes $f(a') + (1+\varepsilon)\mathrm{d}(a', b')$. When either a' or b' was inserted, if the parent was adjacent to the other point, the algorithm updates $g[b']$ guaranteeing that

$$g[b'] \le f(a') + \mathrm{d}(a', b') \le f(a') + (1+\varepsilon)\mathrm{d}(a', b')$$

This easily satisfies the invariant, so we may assume that there is no such edge. So, at some earlier time there was an edge $a \sim b$ that was pruned away where a is an ancestor of a' and b is an ancestor of b'. Let r_0 and g_0 denote the values of r and g when the edge was pruned. When each point of B is inserted, the value of g is initialized by the value of the parent plus its distance and can only get smaller after that. So,

$$g[b'] \le g_0[b] + r_0[b].$$

By the pruning condition, the definition of Lipschitz, and the triangle inequality, we have

$$\begin{aligned} g_0[b] + r_0[b] &\le (f(a) - r_0[a]) \\ &\quad + (1+\varepsilon)(\mathrm{d}(a,b) - r_0[a] - r_0[b]) \\ &\le f(a') + (1+\varepsilon)\mathrm{d}(a', b'). \end{aligned}$$

Thus, the invariant holds. □

4.4 Running Time Analysis

Degree Bounds. The key to efficient dual-tree algorithms is a bound on the degree of the viability graph. If there are only a constant number of neighbors of any point, then each new point only requires constant time.

For the following lemmas, let s be the distance from q to its parent in an iteration of the main loop.

Lemma 7 *Then, for all inserted points p we have $r[p] \le \frac{s}{1-\alpha}$.*

Proof. The value of $r[p]$ is the length of a path in the tree. The first edge in the path has length at most s. Each subsequent edge shrinks by a factor of α. So,

$$r[p] \le s + \alpha s + \alpha^2 s + \cdots \le s\left(\sum_{i=0}^{\infty} \alpha^i\right) = \frac{s}{1-\alpha}.$$

□

Lemma 8 *If $a \sim b$ is not pruned, then*

$$\mathrm{d}(a,b) \le \frac{2 + 4/\varepsilon}{1-\alpha} s.$$

Proof. Because the edge was not pruned, we have

$$g[b] + r[b] > f(a) - r[a] + (1+\varepsilon)(\mathrm{d}(a,b) - r[a] - r[b]).$$

Rearranging this pruning condition, we get

$$\varepsilon \mathrm{d}(a,b) \le (g[b] - (f(a) + \mathrm{d}(a,b))) + (2+\varepsilon)(r[a] + r[b]).$$

Observe that $g[b] \le f(a) + \mathrm{d}(a,b)$ because otherwise, we would have updated $g[b]$ at the time the edge was added. So, the inequality above simplifies to

$$\varepsilon \mathrm{d}(a,b) \le (2+\varepsilon)(r[a] + r[b]).$$

So, by Lemma 7, we have

$$\mathrm{d}(a,b) \le \frac{2+\varepsilon}{\varepsilon} \frac{2s}{1-\alpha} = \frac{2 + 4/\varepsilon}{1-\alpha} s.$$

□

Theorem 9 *Let A and B be subsets of a doubling metric space that have already been processed into β-approximate greedy permutations with an α-scaling parent mapping. Then* `lipschitz_extend(A, B, f)` *runs in $(2 + 1/\varepsilon)^{O(d)} n$ time.*

Proof. The main loop iterates once for each input point. In each iteration, it loops over the neighbors of a point. It will suffice to show that the number of unpruned edges incident to any point is always at most constant. At any iteration, let s be the last insertion distance. Then, any upruned edges $a \sim b$ will satisfy $\mathrm{d}(a,b) \le \frac{2+4/\varepsilon}{1-\alpha} s$ by Lemma 8. Therefore, the diamter of the set of neighbors is at most twice this value. Moreover, any pair of neighbors are s/β-separated by Lemma 4. Finally, Lemma 1 implies that there are at most $\left(\frac{4\beta(2+4/\varepsilon)}{1-\alpha}\right)^d$ neighbors. Assuming reasonable choices like $\alpha = 2/3$ and $\beta = 2$, this yields a degree bound of $(48(1 + 2/\varepsilon))^d = (2 + 1/\varepsilon)^{O(d)}$. This means that we can insert a point in $(2 + 1/\varepsilon)^{O(d)}$ time, giving the desired running time for all $2n$ points. □

References

[1] O. Chubet, D. Sheehy, and S. Sheth. Simple construction of greedy trees and greedy permutations, 2024.

[2] O. A. Chubet, K. P. Gardner, and D. R. Sheehy. A theory of sub-barcodes. In *41st International Symposium on Computational Geometry (SoCG 2025)*, 2025.

[3] O. A. Chubet, P. M. Parikh, D. R. Sheehy, and S. S. Sheth. Approximating the directed hausdorff distance. In *Proceedings of the 35th Canadian Conference on Computational Geometry*, 2023.

[4] R. R. Curtin. *Improving Dual-Tree Algorithms.* PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2016.

[5] S. Har-Peled and M. Mendel. Fast construction of nets in low dimensional metrics, and their applications. *SIAM Journal on Computing*, 35(5):1148–1184, 2006.

[6] R. Krauthgamer and J. R. Lee. Navigating nets: Simple algorithms for proximity search. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 2004.