

Approximate Nearest-Neighbors Using TCQ-Trees

Minya Dai*
UC Davis

Nina Amenta†
UC Davis

ABSTRACT

We present a quad-tree variant that requires $O(n)$ space, to answer low-dimensional approximate nearest-neighbor queries and approximate k -nearest-neighbor queries in $O(d \log h)$ time, where h is the height of the standard quad-tree on the input data. For most “realistic” input data, $h = O(\log n)$, implying that this algorithm provides a log-logarithmic query time using linear space. Achieving the query speed requires a new query-processing algorithm called the projection-based method. Our data structure is dynamic: insertions and deletions can be done in $O(d \log h)$ time as well, and it is fairly easy to implement.

1. INTRODUCTION

In the *nearest-neighbor (NN) problem*, given a bounded input point set P in R^d , and a query point q , we need to return the nearest point $p^* \in P$ to q ; in the *k -nearest-neighbor (kNN) problem*, we return the k nearest points in P . For the exact NN problem in dimension $d > 2$, there is no known solution using near-linear space but achieving poly-logarithmic query time. Better results are available for the *approximate nearest-neighbor (ANN) problem*, in which, given $\epsilon > 0$, we return any point $p \in P$ such that $d(q, p) \leq (1 + \epsilon)d(q, p^*)$. Similarly, in the *approximate k -nearest neighbor ($kANN$) problem* we return a sequence of k points where the i^{th} point is no farther than $(1 + \epsilon)$ times the i^{th} exact nearest neighbor.

Our work is particularly motivated by the recent work in *point-based graphics*, in which surfaces are represented by point clouds. The appeal of using points to represent a surface is that they can be easily manipulated, so it is important to have a dynamic data structure. Although kd -trees are considered effective in practice for answering nearest neighbor queries, it seems difficult to make them dynamic. Simple grids are trivially dynamic and can be implemented in $O(n)$ space but their worst-case query time can also be linear. Arya et al. [1] introduced the BBD-tree, which achieves $O(d \log n)$ query time with $O(n)$ space. Various data structures were proposed afterwards which aim at ϵ - and d -dependency reduction. However, most of these tree structures are constructed to be heavily dependent on the data distribution, which makes point insertion and deletion non-trivial. Quad-trees are an appealing alternative, and Eppstein et al. [4] introduced the skip quad-tree which is optimal in time and space and handles insertions and deletions efficiently.

We observe that if P consists of points from any distribution of bounded density, $h = O(\log n)$. All of the real or simulated [1] input distributions used for testing ANN algorithms, as well the practical inputs we are interested in, have this property. Chan [2] addressed this special case by reducing the problem to $d + 2$ list maintenance

problems in 1D, using a space filling curve closely related to quad-trees. This is a dynamic solution since the 1D list problems can be made dynamic using van Emde Boas trees. Our data structure is another solution in this special case, more directly based on quad-trees but achieving the same bounds. It avoids the constant-factor duplication of Chan’s method and trivially supports traditional tree operations, such as range queries and surface reconstruction.

We call our structure the TCQ-tree. We use hashing to allow binary search along tree paths, and we modify the usual area-expanding tree search algorithm to maintain the explored area’s *boundary* rather than an explicit list of cells. These two innovations allow us to implement the traditional ANN algorithm, using a quad-tree, with $O(d \log h)$ query time and linear space.

2. NATURAL AND COMPRESSED QUAD-TREES

We refer to “quad-trees” regardless of the dimension. In R^d , assuming all points lie in a unit cube, a *natural* quad-tree is constructed by recursively subdividing each bounding cubical cell into 2^d equal-sized d -dimensional cells, until the number of points in each cell has a bounded cardinality. This regular subdivision enables direct access to tree nodes: we use a hash table to store tree nodes, instead of a pointer-based structure. Each node c in the natural tree has a unique *ID*: a 2-tuple $(level, index)$. The *level* field contains c ’s depth in the tree, and the *index* field is a d -tuple, giving c ’s location in the virtual c -granularity grid. The hash table uses this ID as the key to store c .

This direct access allows us to locate the leaf cell which contains a given point in $O(d \log h)$ time; this simple idea is the main tool of this paper. Given point $p = (x_1, \dots, x_d)$ and level l , the index of the cell containing p is $(\lfloor x_1 \cdot 2^l \rfloor, \dots, \lfloor x_d \cdot 2^l \rfloor)$, assuming all $0 \leq x_i < 1$. The cell containing p at any level, if it exists, can thus be accessed in $O(1)$ time through hashing. To locate the leaf containing p , we “round” h up to a power of two: $h' = 2^{\lceil \log(h+1) \rceil}$, and then perform a binary search on the level range $0 \leq l < h'$ checking at each step for the existence of the cell containing p at some level. Therefore, the depth-wise binary search requires $O(d \log h)$ leaf location time.

The size of a natural quad-tree can reach $\Theta(2^d nh)$, thus we want to compress it. In the natural tree, we define a *phantom node* as an internal node with only one occupied child and a *real node* as either an occupied leaf or an internal node with at least two occupied children. To compress the natural tree, we remove all empty leaves and phantom nodes (replaced with edges connecting the real nodes), leaving real nodes only. This results in the *compressed tree*, with $\leq 2n - 1$ nodes. using only $O(n)$ space. However, binary search cannot be performed in the compressed tree the same way as in the natural tree, because phantom nodes are not stored. To organize the irregular number of children in the parent node, we adopt a hash-table based mechanism [3].

Each cell c of the natural quad-tree is associated with a *representative point* $p_c \in P$. If c is occupied, a data point it contains is chosen

*mdai@ucdavis.edu. Computer Science Department, University of California, One Shields Ave, Davis, CA 95616. Fax 1-530-752-5767. Supported by NSF CCF-0093378.

†amenta@cs.ucdavis.edu. Computer Science Department, University of California, One Shields Ave, Davis, CA 95616. Fax 1-530-752-5767. Supported by NSF CCF-0093378, NSF CCF-0331736

as p_c ; otherwise, a data point in its parent cell (which must be occupied) is chosen as p_c . To help with point insertion and deletion, we ensure that each point represents at most two real nodes by a simple bottom-up submission procedure [3].

3. TRACING-COMPRESSED QUAD-TREES

In a *tracing-compressed quad-tree* (TCQ-tree) we store some extra *tracing nodes* to allow us to do binary search in a compressed tree. Here we explain how to do this with a small expansion in space; see [3] for the further reduction to $O(n)$ space. For each real node c , we record the set S_c of its *tracing nodes* - c 's ancestor nodes visited in the binary search for c . "Rounding" the tree depth up to a power of 2 ensures that S_c only depends on c and $|S_c| \leq \lceil \log l_c \rceil$.

With $\leq 2n$ real nodes, each of which has $\leq \lceil \log h \rceil$ tracing nodes, the size of the tracing node set is $O(n \log h)$. Counting the compressed tree with $O(n)$ size, the TCQ-tree costs $O(n \log h)$ space. The binary search on a TCQ-tree for each real node takes $O(d \log h)$ time. For a given point, the TCQ-tree reports the smallest real node containing it, based on which the corresponding natural leaf can be calculated in $O(d)$ time [3]. Thus, the TCQ-tree has the same leaf location complexity ($O(d \log h)$) as the natural tree.

The construction of a TCQ-tree is a simple top-down procedure [3]. Each point insertion or deletion affects no more than four tree nodes, which results in $O(d \log h)$ complexity [3].

4. APPROXIMATE NEAREST NEIGHBORS

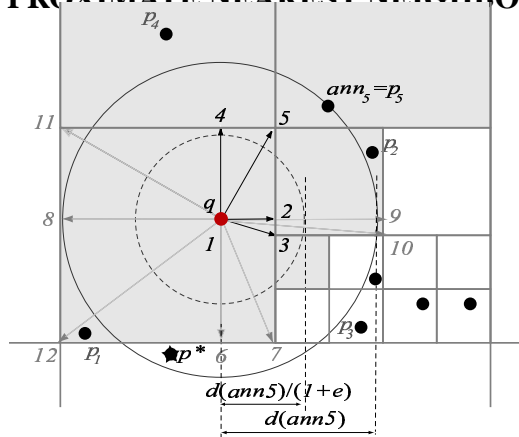


Figure 1: ANN algorithm overview. p^* is the exact NN. Five leaf cells (the grey cells) of the quadtree are checked in order of distance from q , before finding the ϵ -ANN ann_5 ; p_1, p_2, p_3, p_4, p_5 , respectively, are their representatives. The initial seed heap has points $\{2,4,6,8\}$, and only the cell containing q is checked ($ann = p_1$). Then we check the next cell. Point 2, the closest seed, is first popped from the heap. Then the cell point 2 is incident to is checked by comparing its representative point (p_2) with the current candidate (update the candidate if necessary, $ann = p_2$). Finally, the projection points $\{3,5,9\}$ are added to the seed heap. And an area-expanding step is done. We repeat it until the approximation inequality is satisfied.

The standard NN/ANN algorithm on a spatial subdivision is to enumerate leaf cells in order of distance to q . We maintain a candidate ANN point p by comparing the representative of each examined leaf to p and maintaining the closest one. To find the exact NN, we stop when we find a point p^* such that all the leaves covering the region within $d(p^*, q)$ of q are empty. To find an ϵ -ANN, we stop when the region within distance $d(p, q)/(1 + \epsilon)$ of q has been examined and found empty, as in Fig. 1. With a pointer-based data structure (ie. BBD trees, kd -trees or standard quad-trees), all unexamined sibling subtrees of visited nodes are saved in a heap, which produces the closest unexamined subtree for the next step.

The standard implementation will not work with the TCQ-tree, as our direct-access structure and binary search avoids identifying many unexamined subtrees. Instead, we locate the nearest unexamined leaf at each step directly. This leaf can be identified as the one incident to the nearest point s to q on ∂X , the boundary of the examined region; when q is in general position, the leaf is unique.

We keep track of s as ∂X evolves. Instead of explicitly maintaining X (or \bar{X} , as in the standard method), we keep a heap of potential nearest points on ∂X called *seeds*. The initial seeds are the $2d$ projections of q onto the facets of its leaf cell. When a new leaf c is examined and added to X , new seeds - the projections of q onto the $< 2d$ facets of c joining ∂X - are added to the heap. Theorem 1 ensures that the seed heap always contains the closet point on ∂X .

THEOREM 1. *At every step of the ANN area-expanding algorithm, i) the correct nearest cell c is added to X , and ii) after the addition of c , the nearest point to q on any new outward parts of ∂X is a seed. (See Theorem 5 in [3] for proof.)*

We also bound the time complexity of an ANN query, by giving an upper bound on the number of cells examined via a standard packing argument.

THEOREM 2. *An ANN query can be answered using a TCQ-tree in time $O(d(\frac{4\sqrt{d}}{\epsilon})^d(\log \frac{d}{\epsilon} + \log h))$, using a priority queue of size $O(d^2(\frac{4\sqrt{d}}{\epsilon})^d)$. (See Theorem 7 in [3] for proof.)*

We extend the area-expanding process for the k ANN problem by maintaining the sequence of the k closest points to q encountered so far in a binary search tree. The search termination condition is $d(ann^k, q) < (1 + \epsilon)d(\partial X, q)$, where ann^k is the k^{th} approximate nearest neighbor candidate. The j^{th} ($1 \leq j \leq k$) closest points in the sequence is a $(1 + \epsilon)$ -proximation to the j^{th} exact NN, which can be proved through a simple induction. The straightforward extension alone, however, does not guarantee optimal k ANN query time. We address the problem by using a more complicated search structure to avoid checking many empty cells unnecessarily. We give a bound on the number of cell locations [3] and derive the following theorem to give the time complexity of the k ANN algorithm.

THEOREM 3. *The k ANN query can be answered using a TCQ-tree in time $O(((\frac{4\sqrt{d}}{\epsilon})^d + k \cdot 2^d)(d \log \frac{d}{\epsilon} + d \log h + \log k))$, using a priority queue of size $O(d^2((\frac{4\sqrt{d}}{\epsilon})^d + k \cdot 2^d))$. (See Theorem 9 in [3] for proof.)*

5. CONCLUSIONS AND OPEN PROBLEMS

The TCQ-tree potentially has other applications such as range query problems. We are also interested in applying our seed-heap-based approach to other non-quad-tree-based data structures, to improve the search efficiency.

6. REFERENCES

- [1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45(6):891–923, 1998.
- [2] Timothy M. Chan. Closest-point problems simplified on the ram. *SODA '02*, pages 472–473, 2002.
- [3] M. Dai and N. Amenta. Approximate nearest-neighbors using tracing compressed quad-trees. *in progress*.
- [4] D. Eppstein, M. T. Goodrich, and J. Z. Sun. The skip quadtree: a simple dynamic data structure for multi-dimensional data. In *SoCG '05*, pages 296–305, 2005.