Implementing a Parallel Dynamic Approximate Nearest Neighbor Search Algorithm

Michael Connor and Piyush Kumar
Department of Computer Science
Florida State University, Tallahassee, FL

ABSTRACT

We describe the implementation of a fast, dynamic, approximate, nearest-neighbor search algorithm that works well in fixed dimensions \((d \leq 5)\), based on sorting points coordinates in Morton (or z-) ordering. Our code scales well on multi-core/cpu shared memory systems. Our implementation is competitive with the best approximate nearest neighbor searching codes available on the web, especially for creating approximate \(k\)-nearest neighbor graphs of a point cloud.

1. Introduction

Nearest neighbor search is a fundamental geometric problem important in variety of applications including data mining, machine learning, pattern recognition, computer vision, graphics, statistics, bioinformatics, and data compression. Applications of the nearest neighbor problem are particularly motivated by problems like reconstruction, visualization, and simplification for geometry processing applications. The advent of new multi-core architectures and their simplification for multi-core/cpu shared memory systems. Our implementation is based on sorting points in Morton ordering, is easy to implement, can beat ANN given enough processors and gives provably correct answers. The preprocessing time of our algorithm is \(O(n \log n)\) and query time is expected \(O(n \log n + e^{1-\epsilon} \log 1/\epsilon)\) for a single approximate nearest neighbor query. If \(m\) points are available in advance for making queries (batched queries), and \(d < p < m\), then the expected query time for a single point reduces to \(\frac{1}{2} O(n \log n + e^{1-\epsilon} \log 1/\epsilon)\).

Our algorithm is implemented in a thread-safe way. The paper is the first to present extensive empirical results on a multi-core/cpu system. Our algorithm has provable logarithmic query time and is fast in practice compared to the best codes available in its class. Also, while the original algorithm only operated on unsigned integer coordinates, support has been added for both signed integer and signed and unsigned floating point coordinates. The paper presents the first dynamic version of an approximate nearest neighbor search data structure that is competitive with ANN in practice. Finally, our timings for \(k\)-nearest neighbor computation are faster than previous codes as \(k\) increases. We can also compute \(k\)-nearest neighbor graphs faster than ANN in most cases.

There are some drawbacks of our approach. Our algorithm is slower than ANN on single processor systems or if at least \(O(p)\) queries are not available in batches. Four or more cores are recommended for our implementation. The algorithm is randomized and the bounds are expected. The expected running time bounds are independent of distributions from which the point cloud is sampled. Finally, the algorithm is only suited for usage in low dimensions \((d \leq 5)\).

2. The Algorithm

Notation: Given a point \(p = \{p_0, p_1, \ldots, p_{d-1}\} \in P\) where each coordinate \(p_i\) is a \(b\)-bit number \(p_{j_k-1}, p_{j_k-2}, \ldots, p_{j_0}\) its z-value \(z(p) = p_{j_0}p_{j_{k-2}} \cdots p_{j_{k-1}} \cdots p_{j_{k-2}} \cdots p_{j_0}p_{j_{k-2}} \cdots p_{j_{k-1}} \cdots p_{j_{k-2}} \cdots p_{j_0}\). The z-value defines a linear ordering of points in multidimensional space and can easily be calculated by bitwise interleaving its coordinate values. We write \(p \leq q\) iff the z-order of \(p\) is less than \(q\). The z-ordering also defines a quad-tree on the point cloud [2].

We use \(B(p, q)\) to define the smallest quad-tree cell that contains \(p\) and \(q\). We use \(p^{*}\) to denote the shifted point \(p + (s, s, \ldots, s)\). \(P^{*} = \{p^{*} | p \in P\}\). \(d(p, q)\) denotes the
Euclidean distance between $p$ and $q$. We use $nn(p, P)$ to represent the nearest neighbor of $p$ in $P$.

**Preprocessing:** Our preprocessing algorithm is simple. Given a point cloud $P = \{p_1, p_2, \ldots, p_n\} \subset \mathbb{N}^d$ we sort $P$ in $z$-order. This step runs in $O(dn \log n)$ time.

**Query Algorithm:** Our query algorithm is presented in Algorithm 1. The query is done in two phases. The first phase invokes a binary search and the second invokes a modified binary search. The first phase locates the query point $q$ in $P$ using a simple binary search. Line 3 of the algorithm linearly scans points around this location to reduce the radius of the approximate nearest neighbor ball. Line 4 checks if the scan in the previous line yielded a nearest neighbor. In case the nearest neighbor has not been found yet, the second phase of the search is invoked (Line 5), which happens to be a modified version of the query algorithm presented in [2] (Line 7-22). Lines 8-12 do a linear scan to find the nearest neighbor in case the number of points is smaller than a pre-specified constant $v$. In line 13, the nearest neighbor radius is updated by computing the distance from the query point to $p_m$, the mid-point of the modified binary search. Line 14 checks if the approximate nearest neighbor ball intersects with the quad-tree box $B(p_m, p_l)$. Line 15-21 completes the binary search by recursing on one or both subsequences of the Morton ordering. The first phase of the search helps the second phase in many ways. It brings the binary search path into the cache. It helps eliminate some of the calls to the second phase (line 5) and it reduces the size of the approximate nearest neighbor ball, which in turn reduces the number of recursive calls to the second phase (lines 14, 17, 20).

**Algorithm 1 Query algorithm**

```
Require: $P$ is sorted in z-order. $q$ and $P$ are shifted. $\epsilon \geq 0$.
1: procedure QUERY
2:    $i \leftarrow$ BinarySearch($q, P$)
3:    $r \leftarrow nn(q, p_{i-\epsilon} \ldots i+\epsilon)$
4:    if $(q^{i-\epsilon} \geq p_{i-\epsilon}$ and $q^{i+\epsilon} \leq p_{i+\epsilon})$ then return $r$
5:    return CSearch(1, $n$)
6: end procedure
7: procedure CSearch($l$, $h$)
8:    if $(h-l) < \nu$ then
9:        $r \leftarrow \min(r, nn(q, p_{l} \ldots p_{h}))$
10:       return
11:    end if
12:    $m \leftarrow (h+l)/2$
13:    $r \leftarrow \min(r, d(q, P_m))$
14:    if $d(q, B(p_m, p_h)) \geq r/(1+\epsilon)$ then return
15:       if $q \leq p_m$ then
16:          CSearch($l$, $m-1$)
17:       else
18:          CSearch($m+1$, $h$)
19:       end if
20:       if $q^{m-1} \geq p_m$ then CSearch($m+1$, $h$)
21:    end if
22: end procedure
```

The algorithm was implemented in a thread-safe manner, allowing multiple simultaneous queries to be made on the same search structure. Also, using a packed memory array, a dynamic version of the algorithm was implemented, giving comparable (within a constant factor) timing results and allowing points to be inserted and deleted.

**3. Empirical Results**

To establish the practicality of our algorithm, we ran our implementation on a number of different data sizes and with point sets sampled from a number of different distributions. The system that we experimented on is a 8-cpu 2.6Ghz AMD Opteron 885 system, each cpu has dual-core capability with 64GB of total DDR memory. Each core has 1MB of L2 cache and 64kb of L1 cache (both data/instruction). Redhat Linux with kernel 2.6.9-34 was running on the system. We used gcc version 3.4.5 for compilation of all our code (with -O3). ANN was compiled using the defaults, and run with the standard search options. The following graphs are representative of experimental results, it shows the ability of our algorithm to compete with ANN.

![Query time versus number of points using different number of processors for exact nearest neighbor searches. Input: up to 1 million input points from uniform distribution.](image)

**4. Conclusion**

In this paper we have presented an empirical analysis of a simple approximate nearest neighbor algorithm using Morton ordering. The method was designed to scale well on multi-core systems, be cache-friendly and have competitive pre-processing and query times with kd-trees in small dimension.

**References**