NEARPT3 — Nearest Point Query in E3 with a Uniform Grid

[Extended Abstract]

W. Randolph Franklin
ECSE Dept, 6026 JEC, RPI, Troy NY 12180
geom@wrfranklin.org, http://wrfranklin.org

1. INTRODUCTION

We present NEARPT3, an algorithm and preliminary implementation to preprocess a large set of fixed points and then perform nearest point queries against them. With fixed and query points drawn from the same distribution, NEARPT3's expected preprocessing and query time are $\Theta(N)$, with a very small constant factor. NEARPT3, designed for large datasets, has been tested on the largest datasets in the Georgia Tech Large Geometric Models Archive, [7]. Therefore, processing tens of millions of points is quite feasible.

The prior art includes various data structures and algorithms for variants of nearest neighbor searching. The cost of a Voronoi diagram, [6], in E^3 is data dependent, amd runs from $\Omega(N \log N)$ to $O(N^2)$ in time and space for preprocessing, with each query costing $\theta(\log N)$. Range trees, [6], cost $\theta(N \log N)$ time to preprocess, with each query also costing $\theta(\log N)$. ANN (Approximate Nearest Neighbors), [2], is a C++ library for approximate and exact nearest neighbor searching in E^d, allowing a variety of metrics, implemented with several different data structures, based on kd-trees and box-decomposition trees. All these algorithms and data structures are more general, hence bigger, than NEARPT3, which is optimized specifically for the L_2 metric in E^3, although its ideas would generalize.

NEARPT3 appears to be the only method that enthusiastically rejects hierarchical data structures and search techniques. Trees and subdivision searching are more robust against adversarially chosen input. However, we believe, based on tests on real data, that they are often suboptimal in practice. This is true even when the real data is moderately unevenly distributed. The extreme data unevenness that would destroy NEARPT3's performance would also force hierarchical data structures to have many levels. In that case, where the hierarchies would then be faster than NEARPT3, though not fast, a shallow hierarchy would perhaps be the least slow.

NEARPT3 has three stages, as follows. The data structure is a uniform grid, [1, 4].

Prepreprocess: This step, which does not depend on the data, need be performed only once. Hence it is excluded from the time statistics, just as the compilation time is also excluded. Indeed, this prepreprocessing could be forced into the C++ compilation step using the template specialization facilities, though that would be silly.

1. Generate the coordinates (x, y, z) of all grid cells with $0 \le x \le y \le z \le R$ for some fixed R.
2. Sort them by $\sqrt{x^2 + y^2 + z^2}$.
3. Pass down the list in order. For each cell c_1, find the last cell, c_2, whose closest point to the origin is at least as close as the farthest point of c_1. Call c_2 the *stop* cell.

   Since the stop cells are monotonically increasing, all this requires only one pass down the cell list.

   The point is that if a point has been found in c_1, we have to continue searching through c_2 to be sure of finding any closer points.
4. Write the sorted list of cells and stop cells to a file.

Preprocess: Here the fixed points are built into the data structure.

1. Compute a uniform grid resolution, G from the number of fixed points, N_f or get it from the user. A reasonable value is $G = r \sqrt[3]{N_f}$, for $1/2 \le r \le 2$.
2. Allocate a uniform grid with one word per cell, to store a count of the number of points in each cell.
3. Read the fixed points, determine which cell of the uniform grid each would fall in, and update the counts.
4. Allocate a ragged array for the uniform grid, with just enough space in each cell for the points in that cell.

   A ragged array contains storage for the points plus a dope vector pointing to the first point of each cell. The total variable storage is one word per cell, plus the storage for the points.
5. Process the fixed points again, computing for a second time the cell that each falls into. This time, store each point in its proper cell.

   The goal is to minimize both the storage used and the number of storage reallocations. Storage reallocations become especially costly as the program's memory working set approaches the computer's available real memory.

   A possible alternative would be to use a linked list for the points in each cell. However, the space

used for the pointers would be significant, and the points in each cell would be scattered throughout the memory, which might reduce the cache performance.

Another alternative would be to use a C++ STL vector, which reallocates its storage as it grows. Our experience finds this to be very suboptimal.

Query: This reports the closest fixed point to a query point.

1. Determine which cell, c, contains the query point.
2. Using the sorted cell list computed in the prepreprocessing step, spiral out from c until a cell with at least one point is found. Often this is c itself.

   For each cell with coordinates (x, y, z) in the sorted cell list, up to 47 other reflected and rotated cells are derived, such as $(-x, z, y)$. If any ordinate is zero, or any two are equal, there will be fewer other cells.

   It would be possible to do this reflection, rotation, and duplicate deletion in the prepreprocessing stage. This would cause a much larger sort cell list. However it would reduce the query time because that code would have fewer conditionals, which should make it more optimizable.
3. Continue spiralling out until c's stop cell to find any closer points, if one exists.

   This spiralling process is conservative since it ignores the location of the query point inside c.

   On the average 200 cells are searched for each query, but checking each cell is very fast.

2. TESTS

NEARPT3's performance is data dependent. An improper choice of input, such as query points that are very far from all the fixed points, will be intolerably slow. Nevertheless, all the data sets tested so far perform quite well, including these:

Data set name	Source	# fixed points	# queries	CPU time, secs
Bunny	GIT	17973	17974	1.9
Bone6	GIT	284818	284818	28
Dragon	GIT	218882	218883	21
Hand	GIT	163661	163662	16
Uniform random	generated	1M	1M	128

The environment is a 2002-vintage IBM T30 Thinkpad laptop computer with 768 MB of memory, a 1600 MHz Pentium 4 Mobile CPU, and Intel's icpc 8.1 C++ compiler, with all optimizations enabled. The times include reading the data and writing the results. These experiments also validate that the cost is linear. The preprocessing cost is $\Theta(N)$. Each query may cost $O(N)$, but typically costs $\Theta(1)$.

NEARPT3's cost is affected by the grid resolution, however values within a factor of two of the optimum typically change the time less than a factor of 2.

3. EXTENSIONS

NEARPT3 could return approximate nearest matches in much less time since the spiral search could stop sooner. In E^d for other d, the cost of searching is exponential in d, as for any search procedure.

NEARPT2 is a simplified version for preprocessing and searching for points in E^2. We tested 1M queries against 1M fixed points, both sets randomly generated, using NEARPT2, CGAL 3.0.1's NEAREST_NEIGHBOR_SEARCHING, [3], and ANN 0.2, [5]. A proper choice of compiler flags would probably speed both CGAL and ANN, but not reduce their storage cost. None of these tests required any data I/O since the input was randomly generated and the output not written. Performing 1M queries against 1M fixed points cost as follows.

Program	Time	Storage
NEARPT2	9.4	46MB
CGAL NNS	41	120MB
ANN	41	128MB

We then tried 10M fixed and 10M query points but CGAL and ANN required too much memory. NEARPT2 used 458MB and 98 seconds.

4. SUMMARY

The general lesson of NEARPT3 is that simple data structures like the uniform grid can be quite efficient in both time and space in E^3.

5. ACKNOWLEGEMENTS

This research was supported by NSF grant CCR-0306502.

6. REFERENCES

[1] Akman, V., W. R. Franklin, M. Kankanhalli, and C. Narayanaswami. Geometric computing and the uniform grid data technique. Computer Aided Design **21**(7), (1989), 410–420.

[2] Arya, S. and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*. 271–280.

[3] CGAL. The CGAL home page. `http://www.cgal.org/`, 2003.

[4] Franklin, W. R. and M. Kankanhalli. Parallel object-space hidden surface removal. In *Proceedings of SIGGRAPH'90*, volume 24. 87–94.

[5] Mount, D. and S. Arya. ANN: library for approximate nearest neighbor searching version 0.2 (beta release). `http://www.cs.umd.edu/~mount/ANN/`, 1998.

[6] Preparata, F. P. and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[7] Turk, G. and B. Mullins. Large geometric models archive, 2003. URL `http://www.cc.gatech.edu/projects/large_models/`.