View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UCT Computer Science Research Document Archive

Accelerating kd-tree searches for all k-nearest neighbours

B. Merry, J. Gain and P. Marais

Department of Computer Science, University of Cape Town
South African Centre for High Performance Computing

Abstract

Finding the k nearest neighbours of each point in a point cloud forms an integral part of many point-cloud processing tasks. One common approach is to build a kd-tree over the points and then iteratively query the k nearest neighbors of each point. We introduce a simple modification to these queries to exploit the coherence between successive points; no changes are required to the kd-tree data structure. The path from the root to the appropriate leaf is updated incrementally, and backtracking is done bottom-up. We show that this can reduce the time to compute the neighbourhood graph of a 3D point cloud by over 10%, and by up to 24% when $k = 1$. The gains scale with the depth of the kd-tree, and the method is suitable for parallel implementation.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Object hierarchies

1. Introduction

The "all k-nearest neighbours problem" (AKNN) can be defined as follows: given a set of points in space and a number k, find the k nearest points to each of the given points. It is a special case of the k-nearest neighbours (KNN) problem, where the input point cloud is also the set of query points. AKNN is a standard tool in point-cloud processing tasks, including density estimation, normal estimation, smoothing, surface reconstruction and others [CK10]. It is computationally intensive and often dominates the execution time of point-cloud processing tasks [SSV07]. We review previous work on the AKNN problem in Section 3.

A kd-tree can be used to solve the AKNN problem by making N independent KNN queries. Section 2 defines kd-trees and outlines our implementation of them. Our contribution, presented in Section 4, is a simple modification to this approach. We process the points in a spatially-coherent order, which allows some information computed in each query to be re-used for the following query. The modification is general and can be combined with other variations of the problem, such as finding approximate nearest neighbours. The results show a significant reduction in the total time.

2. kd-Trees

A kd-tree is a tree structure where each node corresponds to a *rectangle*: in d-dimensional space, a rectangle is the product of d closed intervals on the coordinate axes. Each internal node has an axis-aligned hyperplane that splits the rectangle; the two sub-rectangles thus formed are associated with the two child nodes. Each point in a point cloud is stored in a leaf whose rectangle contains it. As there is a large body of literature on kd-trees, we will not attempt to review it here. The interested reader is referred to Elseberg et al. [EMSN12] for a comparison of several kd-tree implementations.

A kd-tree can be used to accelerate k-nearest neighbour queries [FBF77], using ball-rectangle intersection tests. Given a query point p and k candidate neighbours, we can be sure that the true k-neighbourhood will be found inside a ball centred on p and passing through the current kth-nearest candidate. When searching for better candidates, a node which does not intersect this ball can be skipped without considering any of its children. Figure 1 shows a 2D example, where $k = 2$. While searching for a neighbour for p, we have identified n_1 and n_2 as candidates. We can ignore any points that lie outside the circle shown, allowing the left subtree of the root to be pruned.

Constructing the tree A kd-tree is constructed recursively, by repeatedly splitting a node into two children and distributing corresponding points to the appropriate children. A splitting rule determines the axis and position of the splitting plane for each internal node. We have used the *sliding*

© The Eurographics Association 2013.

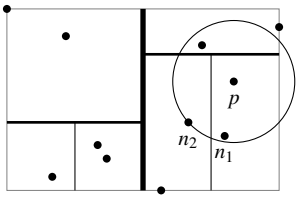

Figure 1: *Search for neighbours in a kd-tree.*

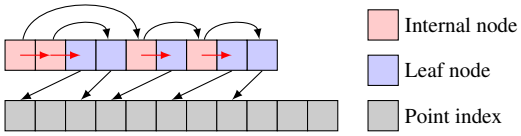

Figure 2: *A flattened kd-tree. The red arrows are implicit pointers to the left child, while other arrows are explicitly encoded array indices. Each leaf node additionally encodes the size of its bucket.*

midpoint rule, which has good theoretical and practical performance for nearest-neighbour searches [MM99].

Elseberg et al. [EMSN12] have obtained high performance in their kd-tree library (`libnabo`) by keeping the nodes as small as possible. We have closely followed their design, where each node is represented with only 8 bytes. The nodes are stored as a flat array, ordered by a pre-order walk (see Figure 2), so it is not necessary to store a pointer to the left child as it will always be adjacent in memory.

To take advantage of a multi-core CPU, we construct the tree in parallel: initially only smaller subtrees are stored in flat arrays, with the top levels of the tree linked together by "super-nodes" (Figure 3) that contain pointers. The separate flat arrays are produced in parallel, then stitched together in a subsequent pass. Further details may be found in our technical report [MGM13].

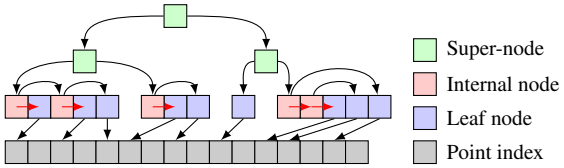

Figure 3: *Hybrid kd-tree. Each* super-node *is allocated separately. Each connected piece of internal and leaf nodes is a separate array that is produced as a serial task.*

Finding nearest neighbours To reduce the number of computations needed to find the distance between the query point and a rectangle, we use incremental distance computations [AM93]. While processing a node N, we store the minimum squared distance between p and N, as well the portion of this squared distance along each axis. When moving to a child of N, we update the per-axis squared distance corresponding to the splitting hyperplane, and use the change in this value to incrementally update the total squared distance. We briefly experimented with visiting leaves strictly in order of increasing distance (*priority search* [AM93]), but found that the overhead of maintaining the priority queue exceeded the gains from examining fewer leaves.

We store candidate neighbours in a sorted array rather than a binary heap as the former is more efficient for small k [EMSN12], which is typically the case for point-cloud processing tasks.

3. Related work

In this section we will focus on the AKNN problem, as the whole field of nearest-neighbour techniques is too broad to be reviewed here. However, we will mention one technique that forms the basis for our contribution. Nüchter et al. [NLH07] use kd-trees to answer nearest-neighbour queries in the Iterated Closest Point (ICP) algorithm. They note that each query is typically quite close to the corresponding query from the previous iteration of the algorithm. To exploit this, they modify the query procedure to return both the closest point and the leaf node that contains it. On the next query, the search is started in this leaf node, and backtracking is implemented explicitly using pointers in the tree to parent nodes, rather than top-down using recursion.

Connor and Kumar [CK10] sort the input points along a space-filling curve, which places points close to many of their neighbours. For each point, they obtain a candidate neighbourhood by testing $O(k)$ elements to either side in the sorted list. The candidate is then refined by finding a conservative range to search and recursively subdividing it, pruning sub-ranges when they provably contain no nearest neighbours. Their implementation also parallelises the queries.

The AKNN problem has also been studied in the context of databases, where it is generalised to the "k-nearest neighbour join": for each point in one set, find the k nearest neighbours in another set. Böhm and Krebs [BK04] and Xia et al. [XLOH04] consider the problem from the perspective of I/O scheduling, but each neighbourhood is still computed independently. Sankaranarayanan et al. [SSV07] improve the search by using the neighbourhood of the previous point as an initial candidate for the neighbourhood of the current point. This gives an upper bound on the search radius, and they further modify the search to avoid re-searching the space occupied by the previous neighbourhood.

© The Eurographics Association 2013.

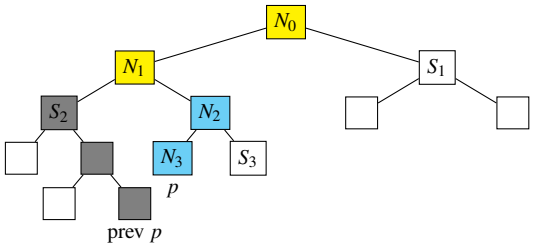

Figure 4: *Primary path update. The gray nodes belong only to the old primary path, and are popped (bottom-up). The blue nodes are then pushed (top-down) until reaching N_m.*

4. Exploiting coherence

Once a kd-tree has been constructed, a naïve approach to solving the all k-nearest neighbours problem is to perform an independent search for each point against the tree. This is sub-optimal, because it discards information determined for one point which can be reused for nearby points. Our approach is based on the work of Nüchter et al. [NLH07], but adapted to the all k-nearest neighbours problem. In particular, our approach does not store parent pointers in the kd-tree.

We iterate over the query points in the kd-tree order, which gives good spatial coherence. Given a query point p, let $P = \{N_0, \ldots, N_m\}$ be the path through the kd-tree from the root (N_0) to the leaf containing p (N_m), as shown in Figure 4. We will call this the *primary path*. Let S_i be the sibling of N_i. If we expand the recursive calls from a standard search that follow the primary path, we find that we first visit N_m and then the subtrees rooted at $S_m, S_{m-1}, \ldots, S_1$. This corresponds to lines 7–13 in Algorithm 1.

To exploit coherence, we compute the primary path incrementally, starting with the primary path for the previous query. We pop nodes that do not contain p until we are left with a prefix of the primary path, and then complete this partial path by walking down the tree as usual (Figure 4). This is shown in lines 1–6. Each node is pushed once and popped once (due to the queries being made in kd-tree order), so this takes amortized $O(1)$ time per query.

We also exit the search early if the rectangle associated with node N_i completely contains the ball centred at p and passing through the kth-nearest candidate — the so-called "ball-within-bounds" test [FBF77] (line 10). This test could also be applied to the naïve search. We found that applying it there reduced performance, while it increased performance in our incremental implementation.

To implement these operations efficiently, we need to associate some extra fields with each node: the range of node indices for the descendants of the node, and the rectangle corresponding to the node. It is not necessary to store these fields in the kd-tree itself: they are maintained only for the primary path.

Finally, we multi-thread our implementation by dividing the buckets into contiguous chunks, and processing each chunk independently. For the first query point in each chunk the primary path is computed from scratch rather than incrementally.

Algorithm 1: Bottom-up backtracking. `FindKNN` is a standard top-down recursive walk. The incremental distance computations are omitted for clarity; refer to our technical report for details [MGM13].

```
   Input: Query point p
   Input: Leaf L containing p
   Input: Primary path P of some point
   Output: Neighbourhood of p
   Output: Primary path of p
1  while L is not a descendant of P.back do
2  |  P.pop();
3  while P.back ≠ L do
4  |  Find child C of P.back containing L;
5  |  Compute rectangle and node range of C;
6  |  P.push(C);
   // P now the primary path of p
7  FindKNN(P.back);
8  foreach N_i in P except the root do // bottom-up
9  |  if N_i completely contains candidate ball then
10 |  |  break;
11 |  d ← dist(p, sibling(N_i));
12 |  if d < current kth-smallest distance then
13 |  |  FindKNN(sibling(P_i));
```

5. Results

Experiments were carried out on an Intel Core i7-2600 (4 cores, 3.4 GHz) with 16 GiB of RAM running Ubuntu 12.04. The code was written in C++ and compiled with GCC 4.6.

Table 1 shows the reduction in total time (including time taken to build the tree) due to our modified search. For comparison, we also show the time taken by the integer version of STANN 0.74 [CK10]. STANN was not able to process the Pisa data set as our test machine has insufficient virtual memory. We have not implemented the scheme of Sankaranarayanan et al. [SSV07], but a comparison with their reported results suggest that their scheme is not competitive for in-core use.

Figure 5 shows the improvement against the number of vertices. In general, larger point clouds benefit more. Large clouds have deeper kd-trees, and so gain the most from eliminating the top-down computation of the primary path.

Our parallelisation is reasonably successful, achieving about a 4.4–4.6× speedup using 8 threads over 4 cores.

© The Eurographics Association 2013.

Table 1: *Data sets and search times with $k = 8$ and 8 threads. The data sets marked with a (*) are range-scanned point clouds, while the other models are reconstructed meshes with the connectivity information removed.* Build *is the time to construct the kd-tree.* Naïve *and* Backtrack *are the times for independent queries and for our method respectively, and* Reduction *is the difference between them. Values in parentheses exclude the build time.* STANN *is the time taken by the STANN library [CK10]. The Armadillo data contains a significant amount of noise and background, while the other range-scanned data sets contain clean data. We also dropped scans from the Armadillo data set that had no registration information.*

Data set	Points ($\times 10^6$)	Build (s)	Naïve (s)	Backtrack (s)	Reduction (%)	STANN (s)
Bunny (*)	0.36	0.02	0.10 (0.08)	0.09 (0.08)	4.6 (5.4)	0.82
Happy Buddha	0.54	0.02	0.14 (0.11)	0.13 (0.10)	6.3 (7.8)	0.97
Turbine Blade	0.88	0.04	0.23 (0.19)	0.21 (0.17)	6.2 (7.5)	1.94
Armadillo (*)	2.93	0.20	0.98 (0.78)	0.88 (0.69)	9.4 (11.7)	6.65
David (2mm)	3.61	0.20	0.99 (0.78)	0.90 (0.70)	8.6 (10.9)	6.89
Lucy	14.03	0.85	3.90 (3.05)	3.47 (2.62)	10.9 (13.9)	27.23
David (1mm)	28.18	1.60	7.67 (6.07)	6.78 (5.18)	11.6 (14.7)	56.42
Pisa (*)	157.43	10.02	48.07 (38.05)	41.81 (31.80)	13.0 (16.4)	—

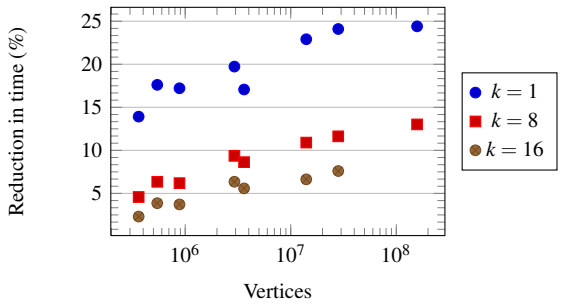

Figure 5: *Improvement over the naïve implementation for the models listed in Table 1 (including build time). Pisa with $k = 16$ is omitted due to lack of memory.*

6. Conclusions

We have presented a modification to a standard kd-tree search that accelerates queries in the all k-nearest neighbours problem, with the greatest improvements when k is small and the point cloud is large. The modification is simple to implement and requires no modifications to the tree structure. It is also extensible to a number of related techniques, such as approximate nearest neighbours, searches within a radius bound, non-Euclidean metrics, user-provided predicates to exclude certain neighbours, and so on. Our implementation supports higher dimensions, but the impact of dimension on performance is unknown.

7. Acknowledgements

The data sets are courtesy of Stanford, Georgia Institute of Technology and the Digital Michelangelo project. Funding was provided by the South African Centre for High Performance Computing.

References

[AM93] ARYA S., MOUNT D. M.: Algorithms for fast vector quantization. In *Data Compression Conference, 1993. DCC '93.* (1993), pp. 381–390. 2

[BK04] BÖHM C., KREBS F.: The k-nearest neighbour join: Turbo charging the KDD process. *Knowl. Inf. Syst. 6*, 6 (November 2004), 728–749. 2

[CK10] CONNOR M., KUMAR P.: Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization and Computer Graphics 16*, 4 (July–Aug 2010), 599–608. 1, 2, 3, 4

[EMSN12] ELSEBERG J., MAGNENAT S., SIEGWART R., NÜCHTER A.: Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. *Journal of Software Engineering for Robotics 3*, 1 (Feb 2012). 1, 2

[FBF77] FRIEDMAN J. H., BENTLEY J. L., FINKEL R. A.: An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw. 3*, 3 (Sep 1977), 209–226. 1, 3

[MGM13] MERRY B., GAIN J., MARAIS P.: *Accelerating kd-tree searches for all k-nearest neighbours*. Tech. Rep. CS13-01-00, Department of Computer Science, University of Cape Town, 2013. 2, 3

[MM99] MANEEWONGVATANA S., MOUNT D. M.: It's okay to be skinny, if your friends are fat. In *Center for Geometric Computing 4th Annual Workshop on Computational Geometry* (1999). 2

[NLH07] NÜCHTER A., LINGEMANN K., HERTZBERG J.: Cached k-d tree search for ICP algorithms. In *Proceedings of the 6th IEEE International Conference on Recent Advances in 3D Digital Imaging and Modeling (3DIM '07)* (August 2007), IEEE Computer Society Press, pp. 419–426. 2, 3

[SSV07] SANKARANARAYANAN J., SAMET H., VARSHNEY A.: A fast all nearest neighbor algorithm for applications involving large point-clouds. *Comput. Graph. 31*, 2 (April 2007), 157–174. 1, 2, 3

[XLOH04] XIA C., LU H., OOI B. C., HU J.: GORDER: an efficient method for KNN join processing. In *Proceedings of the Thirtieth international conference on Very large data bases* (2004), VLDB '04, VLDB Endowment, pp. 756–767. 2

© The Eurographics Association 2013.