Nearest Neighbor Queries using Kd-Trees

Eva Nolan

Introduction

A nearest neighbor query, given a target point and a set of points, identifies the point in the set which has the shortest euclidean distance from the target point. Such queries have many applications including uses in Artificial Intelligence. For example, nearest neighbor queries can be used for data classification AI programs. After being trained on a set of preclassified data, a k-nearest neighbor based classification algorithm will look at the classification of the k nearest datapoints to a new data point that needs to be classified, and can use the most common classification of its k neighbors to predict the classification of the target [4]. Another use of nearest neighbor queries is in geographic searches. Such queries can solve a "restaurants near me" map search by employing a k nearest neighbor query to provide the k restaurants nearest to a person's current location.

The Algorithm

This query can clearly be solved in linear time by calculating the distance between each point and the target. However, for large data sets we hope to query in sublinear time. While many techniques have been developed for this, this paper explores the use of k-d trees for solving nearest neighbor queries.

A k-d trees is a data structure that can be used to store muilti-dimensional points sorted by all dimensions in constant space (as each point is stored at most twice in the tree). To query for the nearest neighbor, a modified range query can be used. Range queries can identify all points in a given k-dimensional range in $O(n^{1-\frac{1}{d}} + k)$ time, where k is the number of points reported, and d is the number of dimensions [1]. When searching for the nearest neighbor, however, it is not known what range will contain that neighbor and no additional points.

A k-d tree approximate nearest neighbor query explores down the k-d tree based on which region the target point lies in. While this technique will generally produce a point near our target, there are instances when the neighbor does not lie in the same region as the target, as is seen in Figure 1. To accommodate for this, we must sometimes search both children of a node. Let ℓ be the line dividing a node's two children's regions. Then at that node in the search, if the ℓ is nearer to the target than the current nearest known point, then there is the chance that a nearer neighbor exists in the region not containing the target. Thus in this case we must search down both children of the node. Otherwise, we need only search the branch of tree who's region contains the target.

Advantages

This method works very well for 2-dimensional nearest neighbor queries. In this case, it can perform queries in average $O(logn)$ time [3] and is an optimal search, in that it always returns the nearest neighbor. Note that in some node formations this runtime is higher due to the need to search both children of many nodes, as is this case in the example shown in figure 2 Due to this efficient runtime and linear storage space, k-d trees are a common data structures used to perform 2-dimensional queries. In the implementation of this algorithm, logarithmic runtime can be observed since as the input size n doubles, the runtime generally grows slowly by a constant, not a multiple. The runtime claim is easy to observe when considering the number of nodes searched by each query. As the input size doubles, the average number of nodes searched increases by approximately 2.5,

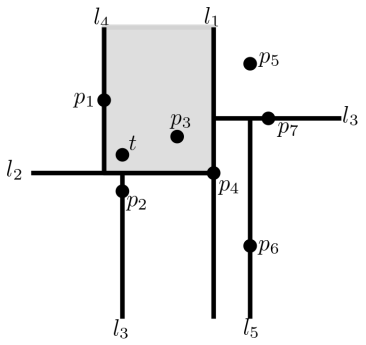

Figure 1: Region diagram for a k-d tree. Point t is target. Nearest neighbor to t is p_2, which lies outside of the shaded region containing t.

this constant addition can be observed even for very large inputs. The exploration of any one node takes constant time, thus the logarithmic number of nodes being visited supports the claimed logarithmic runtime. Runtimes and node exploration counts can be viewed in Table 1.

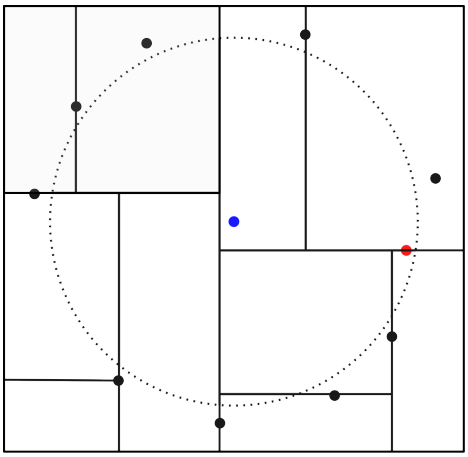

Figure 2: Instance on greater than logarithmic query time. All regions but the lower left must be explored, as they are all within the d of the the target (blue), where d is the distance between target and its nearest neighbor (red), as can be seen by the dotted circle of radius d.

Disadvantages

The biggest drawback of using k-d trees for nearest neighbor queries is their inefficiency when querying higher dimensional spaces. In higher dimensions, this search can approach linear time, in which case brute-force algorithms perform a simpler search with the same accuracy and similar runtime[3]. Due to this massive drawback, alternative methods for nearest neighbor queries are a large field of research. Some research has been conducted on how to alter k-d tree queries for improved runtime performance. Many methods recently developed solve an approximate nearest neighbor query instead, as solving the problem sub-optimally can lower computational costs.

For approximate nearest neighbor queries, a subset of the points are used. Obviously, in searching only a subset of the points, the exact nearest neighbor may be excluded. With careful selection

Table 1: Correlation Between Input Size n, Number of Nodes Visited, and Runtime

n	nodes visited (mean over 100 tests)	runtime (total microseconds over 100 tests)
1	1.0000	1030
2	2.4900	488
4	5.0430	0
8	7.3667	498
16	10.1522	1720
32	14.2903	885
64	15.4211	953
128	18.4536	0
256	21.8351	496
512	26.5876	1028
1024	24.2371	496
2048	33.0000	2777
4096	28.7300	2492
8192	28.1800	3470
16384	35.4100	3741

of the subset, however, it can be guarunteed that the approximate nearest neighbor found will be no more than c further from the target point than its exact nearest neighbor[3], where c is a value predetermined depending on desired efficiency. Other research uses alternate data structures, including randomized partition trees[2], which split on randomly directed lines instead of the axis-parallel lines used by the k-d tree. While these approaches are computationally less expensive, their implementations are more complicated, meaning k-d trees are a good data structure when runtime is not a concern or when a low-dimension search space is being used.

References

[1] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*, chapter 5, pages 99–105. Springer, 3 edition, 2008.

[2] S. Dusgupta and K. Sinha. Randomized ppartition trees for exact nearest neighbor search. In *arcXiv*, volume 1, 2018.

[3] P. Ram and K. Sinha. Revisiting kd-tree for nearest neighbor search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*, KDD '19, page 1378–1388. Association for Computing Machinery, 2019.

[4] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, chapter 19, page 687. Pearson, 4 edition, 2020.