Optimizing Nearest Neighbour Search

Peter Prescott

2021

The Problem

Given a set of randomly generated latitude/longitude points on the surface of the globe, the task was to efficently find, for each point, the index of the nearest neighbour and the distance (km) between that neighbour and the point. The relevant `haversine` formula for finding great-circle distances between points on the surface of the sphere is provided, as is an inefficient `slow` function which generates the correct solutions by iterating through each point (`i`), and then again iterating (`j`) through all the points to find the distance between point `i` and point `j`, recording the relevant index and value if it is the nearest neighbour so far. This is memory efficient, but in time it scales like $\mathcal{O}(n^2)$.

A trivial and slightly `less_slow` improvement would be to only iterate through those points for which we have already calculated the distance, thus reducing the time to some multiple of $\frac{n}{2}(n-1)$; but this is still $\mathcal{O}(n^2)$.

What is needed is some sort of *spatial index* to make it possible to efficiently find the nearest neighbour without calculating distances between points which are obviously far away from each other.

Constructing a KD-Tree

A quick search on Wikipedia (2021) revealed that the K-D Tree is one such solution. The basic idea is fairly simple: we create a binary tree by cycling through the dimensions of our space, and partitioning our points on the median point as sorted by the points' values in that dimension, and recursively building branching sub-trees on either side of the partition node. We can then search for the nearest neighbour by using the tree to guide us towards the desired point. Skrodzki (2019) gives the proof that the expected time taken by a single nearest neighbour search on a kd-tree is $\mathcal{O}(log(n))$ – and therefore $\mathcal{O}(nlog(n))$ for the entire dataset.

There are various implementations freely licensed: I found those by VanderPlas et al. (2012) and Tsoding (2017) particularly helpful.

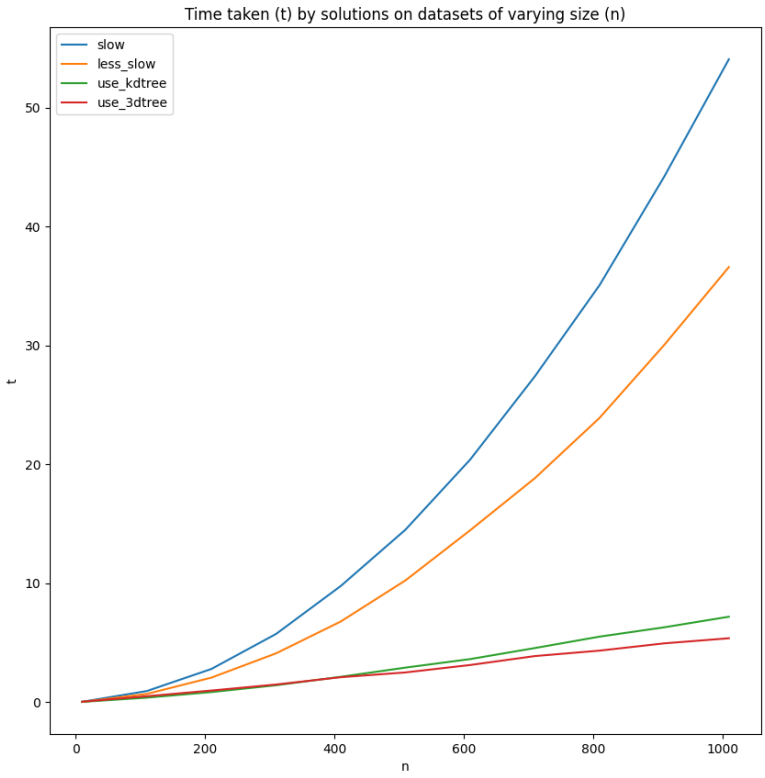

Figure 1: Time taken by Algorithms for Datasets of Different Sizes

However, in spite of the availability of code demonstrating the general concept, implementing a two dimensional kd-tree on the surface of the globe presents a slightly different challenge.

Firstly, the curvature of the earth complicates matters. Although we can treat the dimensions of longitude and latitude as our two dimensions on which we alternately partition our dataset (W., 2015), the shortest distance between a point and its bounding line of longitude is a more subtle matter than the trivial question of finding distances between points and dimensional hyperplanes in Euclidean space.

Secondly, the wrapping of the earth means that if our point or its nearest neighbour is close to extreme ranges of longitude or latitude, our kd-tree will direct us to the wrong solution. The solution I implemented only returns about 75% of the correct answers. Scheidegger (2013) suggests some possible solutions to this, including a 'multicover' of extra wrapped duplicate points surrounding the primary dataset.

But before I had managed to implement this, it occurred to me that it would be conceptually simpler and computationally more efficient to work in three-dimensional Euclidean space, as the symmetry and gentle curvature of the sphere means that a point's nearest neighbour on the surface of the sphere is the same as in 3-D Euclidean space.

When I implemented this, I found that my solution was still only successful for 90% or so of the points in the dataset. On inspection, it turned out that my algorithm was not successfully handling the case where a point reached the branch of which the node was itself. The examples I had based my code on assumed that the point for which one was finding a nearest neighbour would not be a node on the tree. This was not the case in my implementation, and I had thought a simple solution simply returned the other point as the 'closest' if ever the point itself was being considered as a candidate for its nearest point. However, this means that the algorithm 'bounces' off a branch on which the point itself is node, and fails to find the correct nearest-neighbour if it is on that branch. At the time of writing I haven't yet debugged this problem, although I think I have correctly identified it.

Table 1: Time (s) taken by Algorithms on Datasets of Different Sizes

Dataset Size	slow	less_slow	use_kdtree	use_3dtree
10	0.03	0.02	0.02	0.03
110	0.93	0.69	0.38	0.49
210	2.79	2.06	0.84	0.98
310	5.74	4.10	1.42	1.48
410	9.77	6.78	2.13	2.10
510	14.50	10.24	2.90	2.50
610	20.39	14.45	3.62	3.12

Dataset Size	slow	less_slow	use_kdtree	use_3dtree
710	27.39	18.82	4.55	3.87
810	35.05	23.89	5.51	4.33
910	44.18	30.03	6.29	4.94
1010	54.05	36.58	7.18	5.37

Comparison of Times

In spite of the bugs remaining in my attempts to implement a kdtree on the sphere, and a 3-d tree in Euclidean space, we can still see how long they take to find solutions. The table of results (Tbl. 1) and accompanying line-plot (Fig. 1) confirm what we know already, that the `slow` solutions grow quadratically, while the kd-tree solutions are much more efficient.

For a dataset of just over 1000 points, the 3-d kd-tree already takes less than a tenth of the time taken by the `slow` function. For a million points the difference would be even more stark.

All the more reason then to debug this code…

Conclusion

I enjoyed working on this task, and look forward to discussing it on Wednesday. My code is openly available at github.com/peterprescott/optimize-nn, structured as a Python package to be `pip install`ed as desired, with PyTest tests to help make it maintainable (though unfortunately as mentioned, the kd-tree algorithms are not yet quite passing their tests).

References

Scheidegger, C. (2013). Nearest Neighbor Algorithm for Circular dimensions. *Cross Validated*. https://stats.stackexchange.com/questions/51908/nearest-neighbor-algorithm-for-circular-dimensions.

Skrodzki, M. (2019). The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time. *arXiv:1903.04936 [Cs]*. Retrieved from http://arxiv.org/abs/1903.04936

Tsoding. (2017). K-d Tree in Python #1 NNS Problem and Parsing SVG.

VanderPlas, J., Connolly, A. J., Ivezić, Ž., & Gray, A. (2012). Introduction to astroML: Machine learning for astrophysics. In *2012 Conference on Intelligent Data Understanding* (pp. 47–54). https://doi.org/10.1109/CIDU.2012.6382200

W., D. (2015). Data structures - Find k nearest neighbors on a sphere. *Computer Science Stack Exchange*. https://cs.stackexchange.com/questions/48128/find-k-nearest-neighbors-on-a-sphere.

Wikipedia. (2021). Nearest neighbor search. *Wikipedia*.