KD-Tree Coding Exercise

Thomas J. Meehan

11/14/16

My implementation of a KD-Tree can be broken down into 3 major functions: building the tree, searching the tree, and performing disk I/O on the tree structure so that it could persist even when the building function closed and be queried indefinitely by subsequent query functions. Due to this persistent nature, the lack of modifying the tree post-construction, and equal magnitude of tree nodes compared to query points (each at 1000 points in the given files) it was clear to me that a slight sacrifice in speed and space utilized during the construction phase would be a very beneficial trade-off long term assuming that it resulted in highly optimized tree-structure that lent itself to efficient transversal and searches once constructed. Because of this, due to the more straightforward and efficient nature of writing a search algorithm assuming that the underlying tree is optimized I spent most of my time researching and working on a tree-construction algorithm that resulted in a highly optimized tree structure that was also constructed in an efficient amount of time and wasn't too complicated to implement in a relatively small time frame.

It seemed clear to me that finding the more or less exact median and using that as the splitting point for an axis division as efficiently as possible would result in the most efficient structure for future search traversals. After seeing the relatively high cost penalty of sorting the list every time I created a new node, my initial reaction was to instead maintain a list of k vectors and preemptively sort each before creating the tree, referring to each when attempting to find the next median. Unfortunately, this only works once, since on the next level of traversal, assuming you've changed axes, you're only working with half of the nodes as you were previously, and since they're randomly scattered across the new axis' median the preemptive sort doesn't appear to do much, at least on the surface. It turns out I was on the right track, but getting the preemptive sort to work requires a very interesting "super key" based sorting approach. An excellent paper (and even source code) describing the approach can be found here. If I were given additional time it would be the first improvement I'd try to implement, however given the limited time-frame and relative complexity of this approach I decided to implement my own approach by utilizing the std function nth_element. When given the middle index (where the median would appear if the list were fully sorted) it performs a partial sort such that the median element is in its proper spot and everything to the left is less than the median, and everything to the right is greater than the median, which is more or less the bare minimum required for the algorithm. More importantly, it accomplishes this in average O(n) time, meaning the entire construction algorithm takes O(n logn) time on average, putting it on part with the preemptive sort function. Unfortunately, due to being based around a variant of quicksort, the worst case performance is O(n^2), dropping the overall performance to (n^2 logn), whereas the preemptive sort algorithm has worst case performance of (dn logn).

As for choosing the axis to split on, my program can take line arguments to specify either a rotating heuristic (O(1) complexity but not as efficient in most cases for optimizing the tree structure) or a range-based heuristic (O(dn) complexity but generally better resultant tree, especially where points are not evenly distributed along all axes). I chose for the default behavior to be range based since the slight speed decrease during construction appears to be worth the better structured tree, especially in the long term where query computation time outweighs construction time. Speaking of which, my search algorithm has an average complexity of O(log n) and worst case of O(n) depending on how well-balanced the tree structure is.

Finally, I'd like to discuss my disk I/O approach. In short, I tried to use serialization libraries like Boost but I had trouble getting them working in a reasonable amount of time, so I instead decided to be a bit creative and create my own recursive parser to read and write the tree data in plain text in O(n) time and space. Definitely not the most elegant solution, but it works and given what I had to work with I have to admit I'm pretty proud of it. Once again, given additional time (and ideally the guidance of someone more familiar with how to properly integrate those libraries) this is definitely something I'd like to improve on and most likely utilize a proper library for.

Other areas for potential improvement include better error/exception handling (particularly in the event of bad user data input), stronger object oriented design, better organized test cases (including the use of frameworks), and optimizing particular edge cases, such as when the same axis is chosen multiple times in a row when a range-based heuristic is used.