Intro:

K-Dimensional trees, or K-D trees, are a type of binary search tree that are designed to organize multi-dimensional points in space. In lecture we learned about several different types of binary trees and k-d trees differ in that they are specifically well suited for higher dimensional datasets. The image you’re seeing is two different methods for modeling how a k-d tree partitions space, which will we explain in more detail shortly.

Applications:

While there are many real-world applications for K-D trees, one particularly useful one is conducting spatial searches for the nearest neighbor to a given point. This idea can be expanded further to analyze various spatial relationships with multiple parameters. This concept is fundamental in machine learning as it’s used frequently within clustering algorithms and finding similar data. A few other notable applications include ray tracing in game development for better rendering, analyzing multi-dimensional biological data such as protein structures and genetic sequences, and feature matching for object recognition.

Code Explanation:

Edge Struct: We created this structure to be used when creating the DOT file. It keeps track of the connections between nodes by comparing the source, destination, and connection type so that the DOT file can correctly demonstrate the path the search takes.

KDTree Class: NearestNodeID keeps track of the current closest node so that it highlights the ultimate choice made by the search.

KDTree Constructor: Initialize NearestNodeID to -1 because 0 would be a relevant value and skew the data.

findNearestHelper: While this method is recursively finding the nearest neighbor to a given target point, it is simultaneously storing the path that was taken to later be used in generating the DOT file. It stores this information using a set containing the start node, the end node, and a label based on the choice made. After determining which axis to split the tree along, the near and far variables are used to determine if the left or right subtree is closer to the target. The if(near) and if(far) conditionals and edges to the tree and mark them visited so that they aren’t visited twice. From here the ‘path’ set now contains every edge that was traversed during the search, this is then used in the DOTfile method.

Insert: begins with a depth of 0 (one node on the tree) the base case for this method is when the current node is a nullptr, insert a new node with the park and id to be inserted. The variable axis is always either a 0 or 1 corresponding to latitude and longitude to determine which value to compare at that level. This if block determines whether the left or right subtree is used for comparison and increases the depth to represent a new level on the tree. It returns node to ensure the newly created node links back to the original tree.

DOTFile: Iterates over each edge in the path to provide the correct output format for a DOT file generator. Additionally, through the use of the nearestNodeID variable, this method highlights the node representing the nearest neighbor in purple. The ‘or’ statement checks if either the source or the destination for that node is connected to the nearest node, and if so, highlights that connection in purple.

DOT slide: if we can go back to our PowerPoint briefly I included an image of what this output looks like. By using node numbers this map shows how the median choice is made at each level and highlighted in green is the nodes that were traversed by the code. We briefly had the node labels be the names from the input file, however, because many of the names were very long, the image became mostly unreadable.

Main bools: quick check to make sure input values are valid