1 Nearest Neighbor Search Problem

The nearest neighbor search problem is a problem of computational geometry whereby we are given a a set of N target points and M search points. For each of the M search points we are to find the closest corresponding point from the set of N target points. In general these points may be contained within an space of arbitrary dimension, but for the purposes of this project we will consider only two dimensions for examples, and three dimensions for the project implementation.

A naive algorithm that solves this problem has an asymptotic complexity of $\Theta(NM)$. The naive algorithm for this problem would simply compare all possible target points to each of the search points and search for the smallest value. However, this problem can be solved more efficiently, particularly for two and three dimensional problems, with an average cost of approximately $O(M \log N)$. One common technique for solving this problem is to use spatial decomposition trees such as oct-trees or kd-trees. We will describe how to solve this problem with kd-trees in the next section.

2 Overview of kd-trees

The kd-tree is a spatial data-structure that provides a simple and elegant way of structuring spatial data that allows for efficient search and lookup. The kd-tree is a binary tree in which each level of the tree represents a division of the data set into two subsets that are on either side of a selected cutting plane. The simplest version of the kd-tree structure selects cutting planes aligned with each of the coordinate vectors successively. For example, for two dimensional sets, such a kd-tree would first divide with a constant “x” dimension plane at the root, then “y” at the next level, then “x’, and so on. If the plane is selected by the median of the current coordinate value, then the resulting kd-tree will be balanced, thus usually kd trees are constructed using median pivots.

Figure 1 shows how one might construct a two-dimensional kd-tree. In this tree we store nodes at each level of the tree in addition to the pivots (although in some implementations only the leaves of the tree would contain nodes). Notice that the cutting planes only extend to where it intersects with a previous cutting plane.

To understand how this data-structure can help with a nearest neighbor search problem, consider the following algorithm. Given some point, and the current best closest point, we can determine if the left or right partition at the current level of the tree intersects with the current circle described by this radius and the given point. If the partition does not intersect, then it is impossible for a closer point to be contained in that branch of the tree, so it is not necessary to search the nodes contained there.

We illustrate performing a nearest neighbor search using a kd-tree in figure 2. The search begins at the root node which also gives the first candidate closest point. First we choose to look at the left side of the tree and find that the circle that could contain a closer point does intersect this region, and so we proceed to search the left side of the tree. In the second step we find a closer match (point #2) and revise our circle that might contain a closer

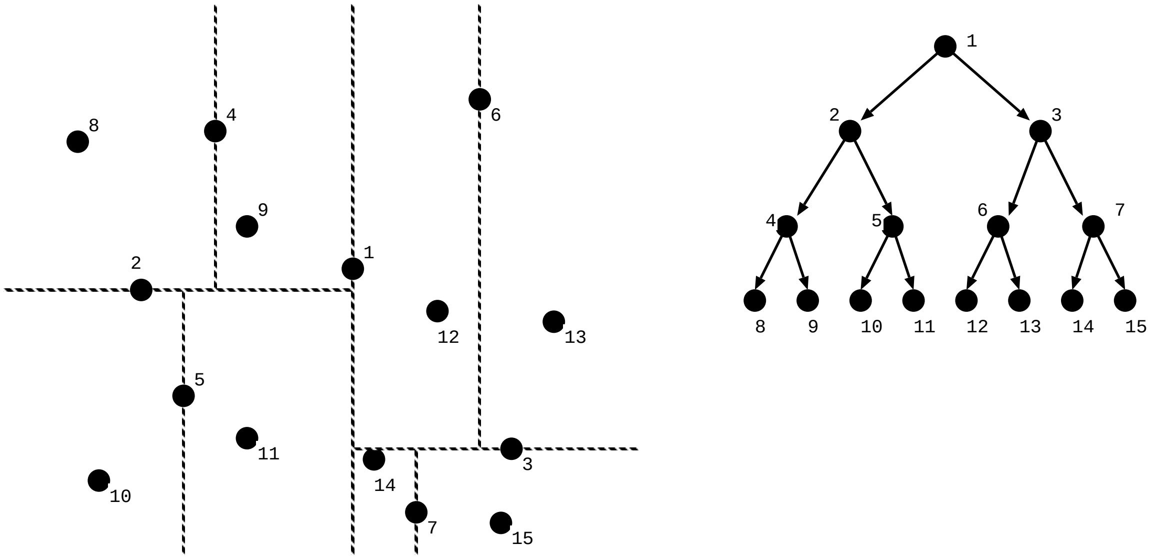


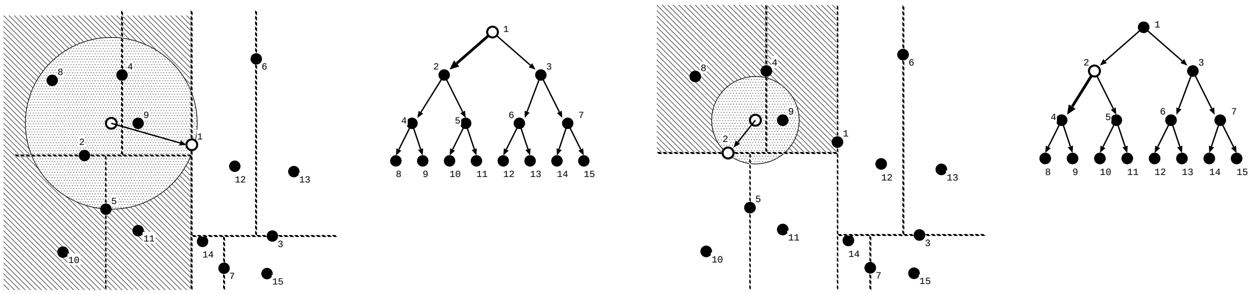
Figure 1: An example of a kd tree

point. Again we consider the left branch of the tree and find that the circle intersects this partition and so we continue searching on the left branch. In step 3, the partition point isn't closer, so we don't need to update the closest point, however the partition for the left branch still contains the circle so we must search it. In step 4 we are at the leaf of the tree, but point #8 is not closer than the current best match so we return to our parent. In step 5 we consider the intersection of our circle with the right branch partition and find that it intersects so we proceed down this arc of the tree. In step 6 we find that the leaf node is closer than the previous estimate, and then return to our parent node. In step 7, we are now check the right partition of this node and find that it doesn't intersect with the circle so we don't search that branch. And finally in step 8 we find that the current best estimate can not be changed by any point in the right half since the current circle does not intersect this partition. Therefore, we know that point #9 is the closest point even though we haven't checked points 5,6,7,10,11,12,13,14, and 15.

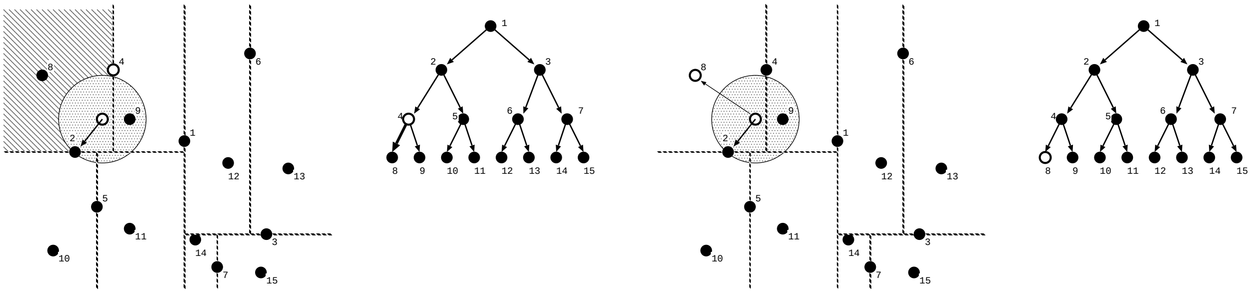
3 Parallel kd-trees

We have several options if we want to develop a parallel kd-tree formulation of the nearest neighbor search problem. The simplest approach would be to collect all of the N target points on all of the processors and have each processor perform a nearest neighbor search using the the serial kd-tree algorithm. This approach has the disadvantage that it would require a total memory of $O(Np)$ thus the approach is not memory scalable.

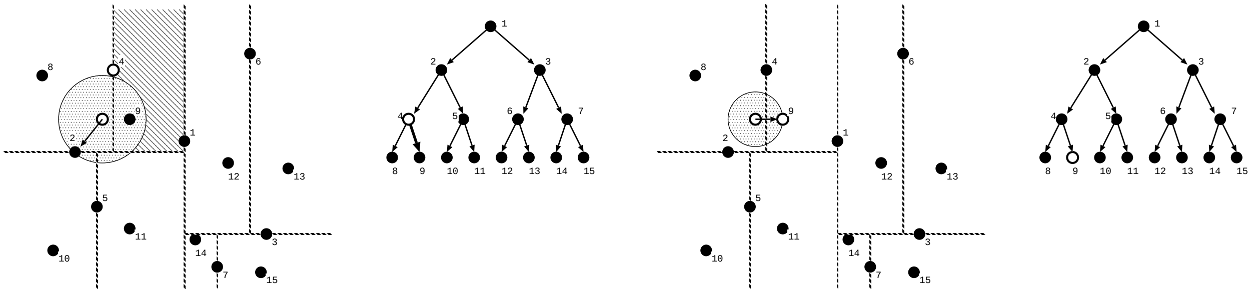
A memory scalable alternative would be to shift the target points among processors in a circular fashion. As the target points are stored on your processor, use them to update



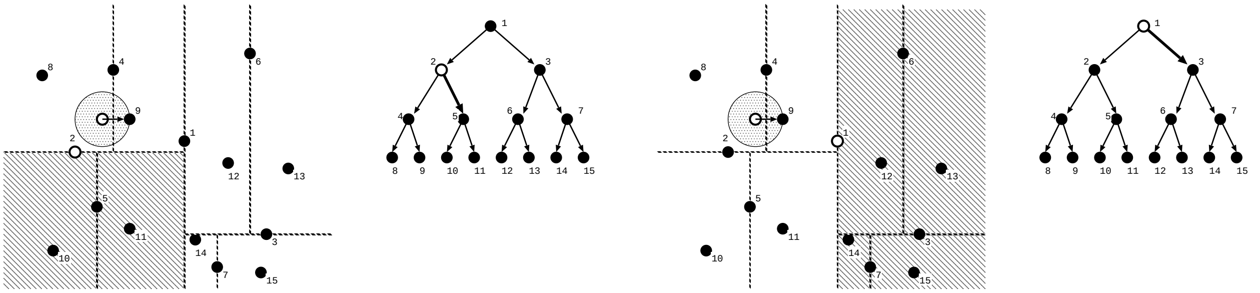
(a) NN search step 1 (b) NN search step 2



(c) NN search step 3 (d) NN search step 4



(e) NN search step 5 (f) NN search step 6



(g) NN search step 7 (h) NN search step 8

Figure 2: Nearest Neighbor Search using kd trees

your local nearest neighbor estimate. After p steps, the algorithm will be finished. However, this approach will not be very scalable as the parallel part of the algorithm mimics the sub-optimal $O(NM)$ algorithm.

The following algorithm can be a scalable approach to performing a parallel k-d tree search. In this algorithm, the search points are partitioned according to an orthogonal recursive bisection (ORB) technique. That is, first the points are divided into two equal subsets by the x coordinate, then the y coordinate, and then the z coordinate. The process repeats until the search points have been partitioned to processors. Then a sample is taken of the target points and shared on all of the processors. Using this sample, the most distant point is determined. One can then use this to compute a bounding box that will contain the all nearest neighbor points. These bounding boxes are shared with other processors and each processor receives the points it needs to search. A k-d tree is constructed from these points and used to find the nearest neighbor for all of the search points.

3.1 Undergraduate Assignment

Undergraduates are to implement the non-memory scalable version and the low scalability version of the nearest neighbors algorithm (the first two algorithms discussed in this document). In addition to implementing and measuring the performance and scalability of these algorithms, undergraduates are expected to perform a scalability analysis of these algorithms. A discussion of the approach, analysis, experimental results, and conclusions should be documented in a project report.

3.2 Graduate Assignment

Graduate students, in addition to performing the undergraduate part of the assignment, will implement the scalable nearest neighbors algorithm. This includes implementing the ORB partitioning algorithm. Note, this algorithm is most easily computed by using a sample sort algorithm. A discussion of the approach, analysis, experimental results, and conclusions should be documented in a project report.