CSC533 Programming Assignment 1 Report

Nate Van

1 Analysis

Both of the algorithms that I implemented are using the basic principle of Heapsort and Randomized-Quicksort, as follows in the book. Because of the additional step of sorting by Euclidean distance, I calculates the distance and stores in a array D first then start the sorting.
This avoids repeating the same calculation, $\theta(n)$. For Heapsort, building the max heap can be visualize as each node representing the corresponding Euclidean distance D_i. So, during max heapify (max_heapify), the algorithm grabs the distance of the current node (largest) $D_largest$, left child D_l, and right child D_r. Then it compares and exchanges the largest and current node as it needs. Same implementation is applied to Randomized-Quicksort by comparing the distance of the pivot D_r and iterated element D_j. Additionally, to compare values with 2 decimal places, I takes the difference of the interested values then compare to the value of epsilon = 0.001.
To make the algorithms simple and organize, I uses C++ template and class for each algorithm. Each algorithm can take in any numberical datatype. They both store in the input array Z, array X, and n number of elements as it instances. Z is a array of 2 elements and X is a 2D array of 2 by n shape. This way all members of the classes have access to the Z, X, and n, which made easy to debug and develop.

2 Results

For testing, I can only genarate $10^5 + 1$ pair random floating point because of hardware limited. Two tests were being ran for each algorithm where the input size of $n = 10^3$ and $n = 10^5$. For the first test, the runtime for Heapsort and Randomize-Quicksort are 1.328ms and 0.577ms respectively. So, Randomize-Quicksort is faster than Heapsort. Even though the Randomize-Quicksort depends on the probability that the random pivot point is not near the edge, the random pivot is generated for each partition. So, as n grows larger, the probability that pivot is closer to the media of the array is higher than near the edge. Thus, it is valid to assume that Randomize-Quicksort is faster than Heapsort on average for large n. Both algorithms produce accurate results by post validation, comparing current and previous elements. For the second test, the runtime for Heapsort and Randomize-Quicksort are 72.456ms and 47.402ms respectively. Similarly, Randomized-Quicksort is faster than Heapsort for $n = 10^5$. Also, both algorithms correctly sorted X. Overall, Heapsort and Randomize-Quicksort correctly calculate the Euclidean distance and sort X accordingly.

3 Source Code

There are two header file heap_sort.h and quick_sort.h and one source file main.cpp. The heap_sort.h contains the heap sort class, and quick_sort.h contains the randomized quick sort class In the main.cpp, it declares array Z, array X, and N number of elements. Main take in a input file and read/store into Z, N, and X, with default file as "test1.txt" if no argument been pass. Main outputs the runtime of Heapsort and Randomized-Quicksort in microseconds to standard out. Also, it outputs validating the accuracy of the algorithms. Then save sorted arrays to files. A additional simple Makefile for quick compile.