CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean
Webpage: `http://www.cs.clemson.edu/~bcdean/`
Handout 18: Homework #4

Fall 2019
MW 2:30-3:45
Daniel 313

1 Shortest Paths Across Campus

This homework tests two main concepts: kd-trees and shortest paths. Starter code is provided in this directory:

`/group/course/cpsc212/f19/hw04/`

In this directory you will also find a large file named `points.txt`, containing aerial LIDAR data describing the elevations of roughly 6 million points on campus. Each line in the file describes a single (x, y, z) point, where x and y increase farther to the east and north, respectively, and z is elevation above sea level. All units are in feet.

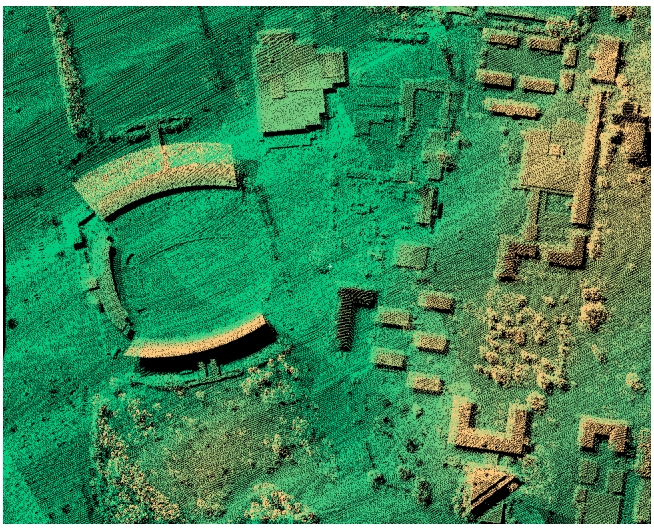

Figure 1: LIDAR elevation data of campus.

The starter code reads this file into a large vector of points for you. It then opens a window that allows you to graphically visualize the data as shown above, scrolling around with the arrow keys, zooming in and out with the plus and minus keys, and quitting by hitting 'q'. The graphical interface is built on top of a library called GLUT, which exists on the lab machines but may take some special configuration on your part if you want to try to make the code compile on your personal computer. Using a lab machine is therefore recommended. If using the virtual.computing web-based interface, you'll need to launch a virtual instance that is GPU-enabled.

To simplify the graphics side of things as much as possible (so you don't need to work with the full GLUT library), the starter code comes with a very simple graphics interface described in the header file `graphics.h`, which contains functions that allow you to change the current color, plot a single pixel, and plot a line segment. The graphics engine is "event driven", meaning that once it is launched from `main()`, it issues callbacks to functions in your code when certain events happen. The `keyhandler` function is called whenever a key is pressed, and the `render` function is called whenever it is necessary to re-render the screen. Right now, the `render` function loops over all the points and plots the ones that are within the boundaries of the current window, color-coding them by elevation and also offsetting their position a bit by elevation (done by the function `get_point_screen_location`) so as to add a 3D effect.

2 First Goal: Build a kd-Tree

Your first task is to build a 2-dimensional kd-tree in the `main()` function from all the points, after they are read in. Each node in the kd-tree stores a single point. You can decide what is the most convenient way to do this — e.g., storing two doubles (x, y) in each node, storing an array of two doubles in each node, or storing an integer index in each node into the `all_points` vector. In any case, be sure each node has enough information to also keep track of the elevation of the point it represents.

Normally, one would insert points in random order into the kd-tree to ensure it comes out balanced, but our points have already been randomly shuffled in the input file, so this is not necessary.

3 Second Goal: k-Nearest Neighbor Queries

Now that you have built a kd-tree, you should write a function that finds the k nearest neighbors of any given (x, y) point (nearest neighbors just in terms of distance based on x and y only, ignoring elevation). Recall that to do this, we should use a branch-and-bound search with pruning as discussed in class. The results of a k-nearest-neighbor query may be most convenient to return in a global structure that is populated as your code recursively searches the kd-tree.

To be able to test if your code is working properly, change the *keypress* handler so that when 'n' is pressed, you search for the 10 nearest neighbors of the point in the middle of the current window. These neighbors should be stored in a global structure so that the `render` method will then highlight these points – for example, we could draw small 'x's over them as follows:

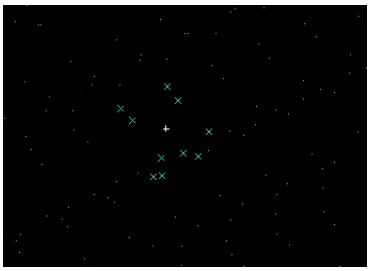

Figure 2: Visualizing the 10 nearest neighbors of a query point.

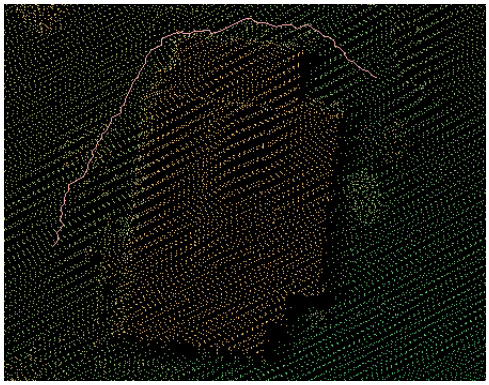

Figure 3: Visualizing a shortest path.

This visual feedback should help you debug your code.

4 Third Goal: Shortest Paths

Observe there are global variables called `source_node` and `destination_node`, initially set to -1. In the keyboard handler, these should be set when you press 's' and 'd' respectively. Using your code for finding nearest neighbors, please modify the keyboard handler so that when 's' or 'd' is pressed, the source or destination node is set to the nearest neighbor of the point currently in the middle of the view window.

After both 's' and 'd' have been pressed, any time either of these keys is pressed we should calculate the shortest path from the source to the destination, in a graph where every point is a node, and each point is connected to its 10 nearest neighbors with edges. You will therefore want to call your k-nearest-neighbor function within Dijkstra's algorithm to enumerate the neighbors of each node. The length of an edge joining two nodes should be the 3D Euclidean distance between the two respective points – e.g., the distance between (x_1, y_1, z_1) and (x_2, y_2, z_2) is just

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}.$$

As a final constraint, we want to make sure we don't find paths that are too *steep*. If any edge has an upward slope of more than 1 (i.e., 45 degrees), then that edge is not to be used. This should prevent your shortest paths from crossing through buildings. The slope when moving from (x_1, y_1, z_1) to (x_2, y_2, z_2) is just the increase in z divided by the distance in x and y:

$$\text{slope} = \frac{z_2 - z_1}{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}}.$$

Any time a shortest path is calculated, print out its length in feet (i.e., the sum of its edge lengths) to the terminal. If there is no valid path (e.g., if all paths are too steep), print out a message indicating this. Note that your shortest path algorithm can terminate immediately once it visits the destination node while scanning outward from the source. This can speed things up considerably if the source and destination are close to each-other. If they are farther away, or if there is no valid path, you can expect your code to run for a much longer amount of time.

Any time you compute a shortest path, you should store it in a global structure that the `render` method should then visualize as a sequence of line segments, as shown in Figure 3.

5 Submission and Grading

Final submissions are due by 11:59pm on the evening of Friday, December 6. No late submissions will be accepted.