Data Structures and Objects
CSIS 3700

Honors Project — Fibonacci Heap

Goal

Implement the FIBONACCI HEAP data structure and compare its performance to the performance of a regular HEAP, where the *decrease key* operation is implemented with a regular *delete* and *insert*.

Background

Heaps are classic data structures that are used to implement a *priority queue* — a container that allows elements to be inserted and removed, but removal always removes the element with the "highest" priority, usually the smallest numeric value.

Traditional heaps assume that an element's priority is static. However, there are common algorithms, such as Dijkstra's shortest path algorithm, where an element's priority may change. Although a traditional heap can be used in these situations, their use may not be better — and in fact may be worse — than a naïve solution.

A Fibonacci heap is a variation of the traditional heap that offers an efficient method to increase a key's priority while maintaining an efficient implementation of the other heap operations. Note that priority can only be increased — by lowering the numeric priority value — and not decreased.

In theory, given a graph with E edges and V vertices, the naïve version of Dijkstra's algorithm requires $\Theta\left(|E| + |V|^2\right)$ operations, using a regular heap requires $\Theta\left((|E| + |V|) \log |V|\right)$ operations, and using a Fibonacci heap requires $\Theta\left(|E| + |V| \log |V|\right)$ operations.

In this project, you will be comparing all three versions to evaulate their performance.

Details

In this project, you should implement three versions of Dijkstra's algorithm:

- A naïve version, storing the fringe in an array;
- A version that stores the fringe in a regular heap;
- A version that stores the fringe in a Fibonacci heap.

Generate a set of random graphs with varying numbers of vertices and edges, with a maximum of $10\,000$ vertices (note: this may increase if the run time is too short). Also generate a set of "worst-case" graphs where every selection causes an improvement for all remaining vertices; again, the maximum number of vertices should be $10\,000$.

Run each version of Dijkstra with each of the test files. For each, obtain the actual run time; if the run time is less than 10 seconds, run the algorithm in a loop until the total time is at least 10 seconds and compute the average time per iteration.

In addition, count the number of times distances are compared anywhere in the program; this will include comparing priorities in the heaps and for every edge in the update phase of Dijkstra.

Compare the theoretical count to both the actual count and the run time to see how well they correllate to one another. Present your results in a brief project summary.

What to turn in

Turn in the source code, output file from the test runs, spreadsheet with the extracted data and the summary.