CSC213 Graph Database Report

Henry Fisher, Maddie Goldman, Matt Murphy

May 15, 2018

1 Project Overview

The goal of this project was to build a database that is effective at storing data that maintains relationships. Based on the data's relational nature and because our interest in we decided that it would make sense to implement a graph structure made up of nodes and edges. This project is able to store connections representing the relationship between students to classes, however the basic structure could easily be modified to store other types of relationships. Nodes can be added through standard input from the user, or from a file storing information about the relationships. When all of the input has been entered, the updated database is written to a file in a format specified in a later section of this report.

2 Design & Implementation

There are four major components of our system. To store data and relationships, we used a graph structure where each data point is a node and each relationship is an edge. To access the graph structure we used a hash table, and to search the graph structure to determine relationships, we used a concurrent breadth-first search. Finally, the I/O component allows users to enter nodes one by one themselves, or read the hash table data from a file. The following sections will provide more details on each component.

2.1 Graph

The graph is built up from two types of nodes: list nodes, and graph nodes. A graph node contains the actual data (a string), the type of node it is (a character representing either student or class), and a list node pointing to the head of a list of that graph node's neighbors (i.e. everything connected to that graph node by an edge). A list node contains a pointer to the current graph node, and a 'next' pointer to the next graph node in the list. When a new entry is added to the database, a new graph node is created with no neighbors. Neighbors must be added one by one using the add neighbors function. Adding nodes, neighbors,

and deleting nodes are functions protected by locks in order to ensure that the concurrent functions can run accurately over the graph structure.

2.2 Hash Table

The hash table is implemented as an array (length 256) of header nodes. The purpose of a header node is to enable concurrency over the hash table. When you want to access a node in the array for any purpose, you need to first acquire the header's lock, then you can see what's inside. Each header node contains a hash node, which is a node created to handle hash table collisions: inside each array entry there will be a linked list of all the hash nodes that belong in that slot. Hash nodes simply contain a pointer to a graph node and a pointer to the next hash node in the structure.

Hashing is done based on the "value" of the graph node. In the student/class case, this is just the name of the student/class. The hash function itself is the djb2 algorithm, copied from a website that is cited in the comment above the function.

2.3 BFS

Given a graph node object, the user may use BFS to return the neighborhood of nodes within some specified distance from the input node. The resulting neighborhood is contained in a hash table, which will be a subset of the original hash table. To account for large neighborhood input distances and densely populated graphs, this function is multi-threaded. Multiple threads draw from a shared node queue and perform breadth-first advancements, appending discovered nodes in the neighborhood to the returned hash table, in a synchronized, thread-safe manner.

2.4 Neighborhood Intersection

A user of this databased may wonder what the overlap is between the neighborhoods of two nodes in the graph. To answer this question, we implement a method to take the intersection between two hash tables. So, for example, if a user of the database wanted to know the direct neighbors shared between two nodes, the user could perform a breadth-first search of neighborhood distance 1, from each of the graph nodes in question, then perform the hash table intersection between those two neighborhoods.

2.5 I/O

The user can add, delete, and search the database by responding to the prompt that appears when running main from the io directory. After entering data into the graph, all of that data is written to a file using the following format:
first line: Type, Value; Type2, Value2; etc subsequent: Type, Value; Type2, Value2; Type3, Value3 where Value has an edge with Value2, and Value3

3 Evaluation

We will measure execution time over graphs of varying densities (i.e. more edges connecting nodes), and varying thread counts. The three plots below show the execution time required to query through 3 neighborhoods using 1, 4, and 8 threads for three different graph structures containing 4 (sparse), 8 (mild), and 16 (dense) classes per student. As seen in the graphs, increasing the thread count increases the execution time. This is likely due to increased message passing between multiple threads, in addition to the relatively high constant time required to create and join each thread. Therefore by this measure, adding threads does not improve the execution time. However, we note that the execution time for each thread count increases as we increase the density of the graph.