CSIS 217 – Advanced Data Structure

Project: Data Structure Exploration in C++

(Spring 2024) – Adam Elhassan

Social Network Connectivity

1. **Overview Of the Project**

   This project mainly aims to find the most direct connections between users within a network. It is implemented through selecting a graph data structures in C++ to represent the network and manipulate its data to achieve desired results.

2. **Design Decisions and Considerations:**

   We decided to simplify the graph structure by storing only user IDs, significantly reducing space complexity. Any additional user data can be retrieved through queries. However, this approach presented the challenge of dealing with empty spaces resulting from deleted users. To address this, we implemented a hashmap to represent the nodes. This decision increased space complexity, but we were able to reduce it by using spp::Sparsehash for the hashmap as it is built on Google's sparse hashmap. This implementation typically requires only one extra byte per entry, compared to approximately 24 bytes with std::unordered_map.

   For the edges, we chose a forward_list due to its efficient scaling with size due to its single-pointer structure. Because the graph is unweighted and undirected, we decided to use a bi-directional BFS technique for the shortest path algorithm, which turned out to be the optimal solution. While both traditional and bidirectional BFS algorithms have a time complexity of O(N), the bidirectional option offers exponentially faster performance as the

graph grows. With a branching factor (B) of 2 and a depth (D) of 10, traditional BFS yields 1024, whereas BiBFS reduces this to 32. (BFS: O(B pow(D)) while BiBFS = O(B pow(D / 2)). Additionally, BiBFS can be threaded.

3. **Justification For Choosing This Data Structure Over Others:**

   Starting this project it was well known that our data would be represented by a graph, but the question was which implementation. We eventually chose the adjacency list representation due to its efficiency, particularly for an undirected and unweighted graph. Unlike the adjacency matrix, it doesn't suffer from redundant data storage in the lower half. Edge lists were also evaluated but it was decided that they are unnecessary given the lack of tangible benefits, especially considering the added complexity they would introduce.

4. **Implementation Details:**
   - **Graph Class Implementation Decisions**

- Structs for vertices to reduce overhead:
- Store edges using std::forward_list: Chosen for scalability with size due to its single-pointer structure and efficient management of user connections without unnecessary memory overhead.

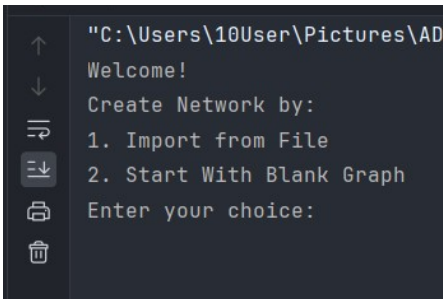
- Employ typedefs for cleaner code structure:
- Implement iterators for seamless iteration over edge and vertex lists:

```
// HashMap used as it was the best way to mitigate the empty slots problem
spp::sparse_hash_map<int, AdjVertex> verticies; // Updated to spp::sparse_hash_map
typedef spp::sparse_hash_map<int, AdjVertex> vertexMap; // Updated typedef
// Cleaner Use for iterators
typedef spp::sparse_hash_map<int, AdjVertex>::iterator vertexIterator; // Updated typedef
typedef std::forward_list<int>::iterator listIterator;
// To Avoid using auto
typedef std::pair<vertexIterator, bool> insertReturn;
// For MultiThreaded Function Calls
typedef spp::sparse_hash_map<int, bool> visitedMap; // Updated to spp::sparse_hash_map
typedef spp::sparse_hash_map<int, int> parentMap; // Updated to spp::sparse_hash_map
```

- Exceptions were not used as speed is a priority 1 and 0 or bools are returned. (1 for false, 0 for true).

  - **Main File Details**

    1) Our Implementation begins with an initial menu. This menu asks the user if they want to create their network from an empty graph or import a graph previously saved in a file.
       a. Option 1 uses function **readFromFile(const std::string& filename)** that retrieves information present in the given file ands adds it to the graph. This function returns 0 if an error occurs, 1 if successful.
       b. Option 2 uses function uses the default graph that is predeclared in the main using the **no arg constructor**.



2) After the user inputs a valid choice, the network is created and the main menu is displayed. This menu contains all the options that the user can choose from to interact with and manipulate the graph. We tried to make the menu more user friendly through terms like “user” for vertex, and “friendships” for edges.
3) **The main function calls 2 functions:**

```
void printFirstMenu();
void printMainMenu();
void mainMenu();
void firstMenu();
int takeAndValidateNumericalUserInput();

int main() {
    firstMenu();
    mainMenu();
    return 0;
}
```

- **firstMenu():** is a function that calls the **printFirstMenu()** function to display the initial menu, and then input the user’s choice and perform the wanted action using a switch and a do while loop.
- **mainMenu():** is a function that calls **printMainMenu()** function to display the main menu, and then inputs the user’s choice and perform the wanted action using a switch and a do while loop as well.

- **takeAndValidateNumericalUserInput():** is a function used by both the menus to take input from the user and validate it, making sure it’s an integer.

4) **The main menu includes 24 different options that use the functions in the graph class:**

```
Main Menu
1.  Display Network
2.  Clear Network
3.  Add User
4.  Delete User
5.  Add Friendship
6.  Delete Friendship
7.  Get Most Popular User
8.  Check If 2 Users Are Connected
9.  Get All Users
10. Get All Friendships
11. Get All Friendships of User
12. Get Degree of Friendship between 2 users
13. Check If Network is complete
14. DFS traversal
15. BFS traversal
16. Check If User Exists
17. Number of Users
18. Number of Friendships
19. Check If Graph Is Regular
20. Get Graph Density
21. Get Average Graph Length
22. Write Current State To File
23. Check If 2 Users Are Friends
0. Exit
```

1. **Display Network:** uses the function **printList()** from the Graph class that prints the adjacency list of this graph.
2. **Clear Network:** uses the function **clear()** that removes everything from the graph.
3. **Add User:** uses the function **addVertex(int id)** that adds a new vertex with the specified ID to the graph.

4. **Delete User:** uses the function **deleteVertex(int id)** that deletes a new vertex with the specified ID to the graph, along with all its associated edges.
5. **Add Friendship:** uses the function **addEdge(int id1, int id2)** that creates a new edge between the vertices with the provided IDs.
6. **Delete Friendship:** uses the function **deleteEdge(int id1, int id2)** that deletes the edge between the vertices with the provided IDs.
7. **Get Most Popular User:** uses the function **getVertexWithMaxDegree()** that returns an integer value representing the ID of the vertex with the maximum number of users.
8. **Check If 2 Users Are Connected:** uses function **isConnected(int id1, int id2)** that returns a boolean value to indicate whether there is a path connecting the vertices represented by the provided IDs.
9. **Get All Users:** uses function **getAllVertices()** that returns a vector of integers representing the IDs of all the vertices in the graph.
10. **Get All Friendships:** uses function **getAllEdges()** which returns a vector of pairs of integers representing the IDs of all edges in the graph.
11. **Get All Friendships of User:** uses function **getUserEdges(int id)** that returns a vector of integers representing the IDs of all vertices that the specified ID is directly connected to in the graph
12. **Get Degree of Friendship between 2 users:** uses the function **shortestPathThreaded(int source, int target).**This function employs a bidirectional BFS traversal with multithreading to efficiently find the shortest path. It returns a vector of integers representing the IDs of the vertices traversed along the shortest path. The size of the vector is also printed as it is the degree of connection between the 2 vertices.

13. Check If Network is complete: uses function **isComplete()** that returns a boolean value to indicate whether the graph is complete or not. In a complete graph, every vertex is directly connected to all other vertices.

14. DFS traversal: takes a user ID and uses **DFS(int startVertex)** that performs a DFS traversal starting from the giving vertex, and returns a vector of integers representing the IDs of the vertices traversed.

15. BFS traversal: takes a user ID and uses **BFS(int startVertex)** that performs a BFS traversal starting from the giving vertex, and returns a vector of integers representing the IDs of the vertices traversed.

16. Check If User Exists: uses function **vertexExist(int id)** that returns a boolean value indicating whether a vertex with the provided ID exists in the graph.

17. Number of Users: uses function **getNbAllVertices()** that returns an integer value representing the number of vertices in the graph. This function uses the nbVerticies datafield in the graph class.

18. Number of Friendships: uses function **getNbAllEdges()** that returns an integer value representing the number of edges in the graph. This function uses the nbEdges datafield in the graph class.

19. Check If Graph Is Regular: uses function **isRegular()** that returns a boolean value indicating whether the graph is regular. A regular graph is characterized by all its vertices having the same degree, meaning they have an equal number of edges connected to them.

20. Get Graph Density: uses function **graphDensity()** that calculates and returns the graph density as a double value. Graph density signifies the ratio of edges present in a graph to the total possible number of edges. Graph density is used to characterize the connectivity of a graph: a high density

means it contains a large number of edges relative to its vertices, while a low density means fewer edges compared to the maximum possible.

21. Get Average Graph Length: uses function **averagePathLength()** that calculates and returns the average path length in the graph as a double value. This measure provides the average distance between pairs of vertices in the graph.

22. Write Current State to File: uses function **writeToFile(const std::string& filename)** to save the current state of the graph to a file, suitable for later retrieval (like option 1 in the initial menu). This function returns 0 if an error occurs, 1 if successful.

23. Check if 2 Users are Friends: uses function **edgeExists(int id1, int id2)** that returns a boolean value indicating whether an edge exists between the 2 provided IDs.

0. **Exit:** exits the program.

- **Graph Class Private Functions**

- **void bfsFromSource(std::queue<int> &s_queue, std::mutex& mtx, std::condition_variable& cv, bool& finished, visitedMap &s_visited, visitedMap &t_visited, parentMap &s_parent, parentMap & t_parent, vertexMap &verticies, int target, int &intersectNode);**
- **void bfsFromTarget(std::queue<int>& t_queue, std::mutex& mtx, std::condition_variable& cv, bool& finished, visitedMap & s_visited, visitedMap & t_visited, parentMap & s_parent, parentMap & t_parent, vertexMap& verticies, int source, int& intersectNode);**

  These 2 Functions are used by the threaded shortest path function.

- **void DFSAux(int startVertex, std::vector<bool>& visited, std::vector<int>& visitedVertices);** This function is a recursive DFS auxiliary function that the public DFS traversal function uses.

    - **More Functions in the Graph Class**

- **Graph(int nbElements):** With arg Constructor that takes an expected number of elements and creates the graph based on that number. This constructor uses the **rehash()** function.
- **Rehash():** this function is used to resize the graph while making sure the preferred load factor is maintained.
- **isEmpty():** returns a Boolean indicating if the graph is empty.

5) Testing And Validation:

To ensure the reliability and correctness of our social network connectivity system, we conducted extensive manual testing procedures covering various aspects of functionality and performance.

Manual Testing:

Manual testing procedures were employed to validate the functionality of core system operations, including user and friendship management, graph traversal, and connectivity analysis. We made test cases to assess the correctness of each function under different scenarios, including edge cases and boundary conditions.

For instance, we tested the addition and deletion of users and friendships, ensuring that these operations were performed correctly and

that the graph structure remained consistent after each modification. Similarly, we evaluated the accuracy of connectivity checks and graph traversal algorithms, verifying that the system produced the expected results in various network configurations.

Our manual testing efforts covered a wide range of scenarios, including:

- Adding and deleting users and friendships.
- Checking connectivity between users.
- Performing graph traversal using depth-first search (DFS) and breadth-first search (BFS) algorithms.
- Calculating the degree of friendship between users.
- Importing and exporting network data from/to files.

6) Challenges Faced:

Throughout the development process, we encountered several challenges that required careful consideration and problem-solving.

- **Vertex Storage:**
  One significant challenge involved managing space complexity while ensuring efficient data access and manipulation. The decision to use a hashmap for node representation helped address this challenge but introduced complexities related to handling empty spaces resulting from deleted users.

- **Library Compatibility:**

Additionally, We directly used Google's sparsehash which after compiling and installing system wide worked fine on linux, but would not compile on windows. After intensive research an email discussing spp (A library built on top of sparsehash) was found in the google archives. Resulting in a smooth transition and refactor.

- **System Setup For Database:**
  It was initially planned to implement the Database connection into this version of the library as an option, and despite it being very simple it would require each user to have the C++ sql connector on their machine even if we were to use a cross—platform library they are built on the connector.

- **Clean Multi threading:**
  Multi-threading BiBFS was very messy code wise at first, especially protecting the shared buffers. After some research OMP was used for multi-threading allowing for cleaner code and safer multi-threading using a tested technology.

- **Deleting Elements From Iterator List:**
  In the delete vertex function delete edge was used at first in order to delete all edges pointing to the vertex being deleted. The program would then crash from the iterator pointing to null due to the elements being deleted from the iterator's list as well. Resetting the iterators each loop iteration would always leave one element left. So, deleting edges of the deleted vertex is done manually then the outgoing list is automatically deleted.

7) **Lessons Learned:**

This project has taught us a lot including but not limited to cleaner C++ code, Iterators, Combining Data-structures for the optimal solution, Time and Space complexity trade-offs and how to reach a balance, problem solving, case analysis, and of course a deep understanding of the algorithms and data-structures used.

8) Future Enhancements

Our future optimization plans include integrating additional features, such as community detection algorithms, to enhance the depth of our social network analysis. These advanced functionalities will enable us to handle more advanced networks, ensuring scalability without compromising performance. Additionally, we recognize the importance of incorporating a database connection. This database must synchronize with the implemented network graph, facilitating efficient data management and providing a robust foundation for further network analysis, such as automatically returning user data rather than ids only as an example.