# **Sequential Breadth-First Search (BFS) in C++ (Class-Based)**

## **1. Overview**

This project provides a sequential Breadth-First Search (BFS) algorithm implemented within a C++ Graph class. The graph uses an adjacency list representation (an array of std::list) to store connections between integer-based nodes.

The primary purpose of this code is to serve as a **sequential benchmark**. The bfs method includes a high-precision timer to measure its execution time. This baseline performance is essential for later comparison against a parallel version of the algorithm.

The main function demonstrates how to use the Graph class by running a small, hard-coded 7-node graph.

## **2. Code Breakdown**

### **class Graph**

This class encapsulates all the logic and data related to the graph.

* **Private Members:**
  + int V: Stores the total number of vertices.
  + list<int> \*l: A pointer to a C-style array of std::list. Each index i in the array holds the adjacency list for vertex i.
* **Public Methods:**
  + Graph(int V): The constructor. It initializes the graph with V vertices and allocates memory for the adjacency list array.
  + addEdge(int u, int v): Adds an undirected edge between vertex u and vertex v.
  + printGraph(): Prints the entire adjacency list representation of the graph to the console.
  + bfs(): This is the core algorithm.
    1. Records the start\_time using chrono::high\_resolution\_clock.
    2. Initializes a queue<int> for the frontier and a vector<bool> vis (size V) to track visited nodes.
    3. Pushes the starting node (hardcoded as 0) onto the queue and marks it as visited.
    4. Begins a while loop that runs as long as the queue is not empty.
    5. Inside the loop, it dequeues a vertex u, prints it, and then iterates through all its neighbors (v) in its adjacency list (l[u]).
    6. If a neighbor v has not been visited (!vis[v]), it's marked as visited (vis[v] = true) and enqueued.
    7. After the loop, it records the end\_time, calculates the duration, and prints the total execution time.

### **main()**

The main() function is the entry point for the program:

1. **Small Graph Example:**
   * A small 7-node graph (g\_small) is created manually by calling addEdge().
   * g\_small.bfs() is called to run the algorithm and print its execution time.
   * g\_small.printGraph() is called to show the structure.
2. **Medium Dataset (Available but not active):**
   * The code contains a *commented-out* section designed to load a larger graph from an external file (medium\_graph.txt).
   * This is intentionally disabled to keep the default execution simple. To test with the larger dataset, you can uncomment this block in main().

## **3. Data File (medium\_graph.txt)**

A data file named medium\_graph.txt is available for testing the medium-sized dataset. This file is **not** read by the program in its current state. If you choose to uncomment the file-loading logic in main(), ensure this file is in the same directory as the executable.

Its format is:

1. **First line:** A single integer V (the total number of vertices).
2. **Subsequent lines:** Pairs of integers u v, each representing one undirected edge.
   * 50
   * 0 10
   * 10 20
   * ...

## **4. How to Compile and Run**

You will need a C++ compiler (like g++).

* + **Compile the code** from your terminal:  
    g++ -o bfs\_program sequential\_bfs.cpp -std=c++11
  + g++: The compiler.
  + -o bfs\_program: Creates an executable file named bfs\_program.
  + sequential\_bfs.cpp: The source file to compile.
  + -std=c++11: Enables C++11 features (for <chrono>).
  + **Run the executable:**./bfs\_program

## **5. Example Output (Default)**

* + --- Small Graph Example ---
  + BFS Traversal starting from vertex 0:
  + 0 1 2 3 4 5 6
  + BFS execution time: 0.00000900 seconds
  + Graph Adjacency List:
  + 0 -> 1, 2,
  + 1 -> 0, 3, 4,
  + 2 -> 0, 5, 6,
  + 3 -> 1,
  + 4 -> 1,
  + 5 -> 2,
  + 6 -> 2,
  + --- Medium Dataset Example ---
  + A medium dataset ('medium\_graph.txt') is available for testing.
  + The code to load it is currently commented out to focus on the small graph.

## **6. Future Parallelization Strategy (OpenMP)**

To parallelize this algorithm for the next phase of the project, we will use a **level-synchronous** strategy with **OpenMP**.

The sequential version explores the graph node-by-node. The parallel version will explore the graph *level-by-level*. The work of processing all nodes *within* a single level will be parallelized.

### **Proposed Algorithm:**

1. **Data Structures:**
   * vector<bool> vis: This will be shared by all threads.
   * vector<int> current\_level: A dynamic array (vector) will replace the queue. This holds all nodes at the current frontier (e.g., Level k).
   * vector<int> next\_level: A temporary vector to store all newly discovered nodes for the *next* frontier (Level k+1).
2. **Parallel Loop:**
   * The outer loop while (!current\_level.empty()) will remain sequential, managing the transition from one level to the next.
   * The core work will be inside a parallel for loop:  
     #pragma omp parallel for
   * for (int i = 0; i < current\_level.size(); ++i) {
   * int u = current\_level[i];
   * // ... process neighbors of u ...
   * }
   * This #pragma tells OpenMP to split the iterations of this for loop among a team of threads.
3. **Synchronization (The Bottleneck):**
   * When a thread processes node u and finds a neighbor v, it must check vis[v].
   * If !vis[v], the thread must:
     1. Set vis[v] = true.
     2. Add v to the *shared* next\_level vector.
   * **This is a critical section.** If two threads find the same new node v at the same time, one might add it while the other is still checking, leading to errors.
   * We must protect this block with a "lock":  
     #pragma omp critical
   * {
   * if (!vis[v]) {
   * vis[v] = true;
   * next\_level.push\_back(v);
   * }
   * }
   * This lock ensures only one thread can check vis and update the lists at a time, preventing race conditions.
4. **Level Transition:**
   * After the parallel loop finishes (all threads are done with current\_level), the main (sequential) thread will set current\_level = next\_level and clear next\_level to prepare for the next iteration.