TEXAS

The University of Texas at Austin

COE 322

12/7/2024

Graph Algorithms

Submitted To: Professor Eijkhout & Susan

University of Texas at Austin

Submitted By:

Names: Orlan Oconer & Dominic Nguyen
EID: ojo366 & dhn454
TACC Username: orlan & dnguyen_3002

Contents

1 Introduction **2**
1.1 Unweighted SSSP . . . 2
1.2 Weighted SSSP . . . 2
1.3 Matrix Operations in Graph Algorithms . . . 2
1.4 Sparse Matrices . . . 2

2 Detailed Presentation **3**
2.1 Headers . . . 3
2.2 C++ Files . . . 5
2.3 CMakeFile . . . 6

3 Experiments **6**

4 Results **7**

5 Conclusion **8**

A Code Appendix **10**
.1 CMakeLists.txt . . . 10
.2 dijkstral.cpp . . . 11
.3 graph_generator.cpp . . . 13
.4 graph_generator.hpp . . . 15
.5 graphmvp.cpp . . . 16
.6 graphmvpdijkstra.cpp . . . 18
.7 graphsparse.cpp . . . 20
.8 queuedijkstra.cpp . . . 22
.9 queuelevel.cpp . . . 24

1 Introduction

Graph algorithms are foundational tools in computer science, enabling solutions to complex problems across a wide range of fields, from logistics and transportation to social network analysis and artificial intelligence. As modern systems grow in scale and complexity, the need for efficient, scalable graph algorithms has become critical. Traditional methods, while robust, often fall short in handling the sheer volume of data generated in applications such as real-time navigation, large-scale simulations, and machine learning.

This project delves into the computational challenges of graph processing by focusing on Single Source Shortest Path (SSSP) algorithms, both unweighted and weighted. We explore innovative computational methods, including the use of matrix operations and sparse data structures, to address scalability concerns. Moreover, by leveraging parallel processing techniques with OpenMP, we aim to enhance algorithm efficiency on large datasets. Through a systematic comparison of traditional and optimized methods, this study not only highlights the trade-offs between time complexity and memory usage but also provides insights into best practices for applying graph algorithms in real-world scenarios. By bridging the gap between theoretical foundations and practical implementations, this project contributes to the development of efficient computational frameworks for the challenges of the modern era.

1.1 Unweighted SSSP

As evident in the naming scheme, unweighted means that every edge is treated equally. The main goal, also evident in SSSP, is to find the shortest path. Starting from the source node, which is 0 because it has no edges yet, it will explore its neighbors and track the distance until all the nodes have been computed. The distance of each node relative to the source node is then saved.

1.2 Weighted SSSP

As the name suggests, the edge has an associated weight this time. The goal is to minimize the system's total weight, trying to connect each node as efficiently as possible. Starting from the source node, all the nodes will be treated equally and a distance "infinity." Using a priority queue, it will visit a node with the closest distance in this queue. After it is processed, it will search for the nearest distance of the current node. Based on the weight of this connecting node, it will update the distance or reshuffle itself in the priority queue. With this method, the weights are given to the nodes with the highest priority, and the search for neighboring nodes will continue without reprocessing the previous nodes.

1.3 Matrix Operations in Graph Algorithms

Matrix operations are connotated with graph algorithms to represent the graph using matrices. This can drastically increase the efficiency of the algorithms. This is where the graphs are implemented as add / mult routines to correspond to the SSSP on the weighted graphs.

1.4 Sparse Matrices

Sparse matrices are important where most of the data is zero or irrelevant. For large graphs, most of the nodes are not connected, so storing non-zero elements becomes memory-efficient. Moreover, it will lead to faster computations, which can drastically increase the SSSP operation time.

2 Detailed Presentation

Now that the fundamentals are fully understood, the inner workings of the code can be explained. Below is the code file tree, which will explain how the files talk and connect to each other.

```
Graph Algorithms
├── CMakeLists.txt
└── src/
    ├── dijkstral.cpp
    ├── queuelevel.cpp
    ├── queuedijkstra.cpp
    ├── graphmvp.cpp
    ├── graphmvpdijkstra.cpp
    ├── graphsparse.cpp
    ├── graph_generator.cpp
    ├── Dag.hpp
    ├── Vector.hpp
    ├── AdjacencyMatrix.hpp
    └── SparseMatrix.hpp
```

2.1 Headers

The headers that were used were Dag.hpp, Vector.hpp, AdjacencyMatrix.hpp, and SparseMatrix.hpp.

- **Dag.hpp**: Represents a Directed Acyclic Graph (DAG), a general graph structure. This structure allows the files to interface with the graphs independently from each other.

```
// Sample DAG structure
class DAG {
public:
    void addEdge(int u, int v); // Adds a directed edge u -> v
    void topologicalSort();    // Performs topological sorting
private:
    std::vector<std::vector<int>> adj; // Adjacency list
};
```

  This header defines a general DAG structure and operations like adding directed edges or performing topological sorting.

- **Vector.hpp**: Contains a vector class that stores distances and information about other nodes. This is particularly useful as many of the .cpp files interact with these distances.

```
// Vector class for storing distances
class Vector {
public:
    std::vector<double> distances;
    void setDistance(int node, double value); // Set distance to a node
    double getDistance(int node) const;      // Get distance to a node
};
```

  This header facilitates interaction with node distances, which is crucial for graph algorithms like shortest path computations.

- **AdjacencyMatrix.hpp**: Responsible for graph representation, storing, and operating on graphs in a dense format.

```
// Dense graph representation using adjacency matrix
class AdjacencyMatrix {
public:
    void addEdge(int u, int v, double weight); // Add weighted edge u -> v
    double getEdgeWeight(int u, int v) const;  // Get weight of edge u -> v
private:
    std::vector<std::vector<double>> matrix; // 2D matrix
};
```

This header enables dense graph representations, useful for algorithms requiring direct matrix operations.

- **SparseMatrix.hpp**: Handles storage of non-zero entries to optimize memory usage. As shown in Section 1.4, this plays a key role in code optimization.

```
// Sparse matrix representation using coordinate format
class SparseMatrix {
public:
    void addEntry(int row, int col, double value); // Add non-zero entry
    double getEntry(int row, int col) const;      // Retrieve entry
private:
    std::map<std::pair<int, int>, double> entries; // Sparse storage
};
```

This header optimizes memory usage by storing only non-zero entries, which is particularly helpful for sparse graphs.

- **Cache.hpp**: Implements caching functionality to optimize operations like shortest path calculations, matrix-vector multiplications, and sparse matrix representations.

```
// Caching shortest paths and other operations
#include <unordered_map>
using CacheKey = std::pair<int, int>;
using CacheValue = std::vector<int>;

struct HashFunction {
    size_t operator()(const CacheKey& key) const {
        return std::hash<int>()(key.first) ^ std::hash<int>()(key.second);
    }
};
```

This header defines caching utilities, enabling significant performance improvements by avoiding redundant computations.

2.2 C++ Files

Below, each cpp file will be explained in its function in conjunction with the graph algorithms and the added caching mechanisms.

- **graph_generator.cpp**: This file creates random graphs, based on the user input of n (the number of nodes) and p (the edge probability).

```
// Generate a random graph with n nodes and edge probability p
void generateGraph(int n, double p) {
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            if ((rand() / double(RAND_MAX)) < p) {
                addEdge(i, j); // Add random edge
            }
        }
    }
}
```

- **dijkstral.cpp**: This file tests Dijkstra's algorithm stored as a DAG. It now includes a caching mechanism to store previously computed shortest paths.

```
// Caching shortest paths in Dijkstra's algorithm
std::unordered_map<std::pair<int, int>, std::vector<int>, HashFunction> pathCache;

CacheValue dijkstraAlgorithm(int source, int target, const DAG& graph) {
    // Compute shortest path
}
```

- **queuelevel.cpp**: Implements the SSSP and includes caching for frequently queried source-target node pairs.

```
// Caching for SSSP results
CacheValue getSSSPWithCache(int source, int target, DAG& graph) {
    // Use pathCache for frequently queried results
}
```

- **queuedijkstra.cpp**: Adds caching to the weighted queue-based system, which helps reduce redundant calculations when processing the same graph multiple times.

```
// Cache-weighted Dijkstra with priority queue
CacheValue weightedDijkstraWithCache(int source, int target, DAG& graph) {
    // Compute and cache results
}
```

- **graphmvp.cpp**: Introduces caching for matrix-vector multiplications during shortest path relaxations.

```
// Caching results of matrix-vector multiplication
std::unordered_map<std::pair<int, int>, std::vector<int>, HashFunction> matrixCache;

MatrixCacheValue matrixVectorMultiplyWithCache(const AdjacencyMatrix& matrix,
                                               const std::vector<int>& vector) {
    // Compute or retrieve cached result
}
```

- **graphmvpdijkstra.cpp**: Uses cached matrix operations to improve performance in matrix-based Dijkstra implementations.

```
// Matrix-based Dijkstra with cached operations
MatrixCacheValue matrixDijkstraWithCache(const AdjacencyMatrix& matrix, int start) {
    // Use caching to optimize
}
```

- **graphsparse.cpp**: Caches sparse matrix representations to avoid recomputing them for the same graph multiple times.

```
// Sparse matrix caching
SparseMatrixCacheValue computeSparseMatrixWithCache(const SparseMatrix& matrix) {
    // Check cache or compute new sparse representation
}
```

2.3 CMakeFile

A CMakeFile was implemented to allow all the files to work and compile with each other. This will allow all the .cpp files to communicate with each other.

```
# Example CMakeLists.txt
cmake_minimum_required(VERSION 3.10)
project(GraphAlgorithms)

add_executable(graph_generator graph_generator.cpp Dag.cpp)
add_executable(dijkstra dijkstral.cpp Dag.cpp Vector.cpp Cache.cpp)
add_executable(queuelevel queuelevel.cpp Dag.cpp Vector.cpp Cache.cpp)
add_executable(queuedijkstra queuedijkstra.cpp Dag.cpp Vector.cpp Cache.cpp)
add_executable(graphmvp graphmvp.cpp AdjacencyMatrix.cpp Cache.cpp)
add_executable(graphmvpdijkstra graphmvpdijkstra.cpp AdjacencyMatrix.cpp Cache.cpp)
add_executable(graphsparse graphsparse.cpp SparseMatrix.cpp Cache.cpp)
```

3 Experiments

For the sake of some of the larger numbers, the graph below is an example where n = 10 and p = 0.1. This is shown in the figure below.

$$\begin{bmatrix} \cdot & \cdot & \cdot & 6 & \cdot & \cdot & \cdot \\ 6 & \cdot & 19 & \cdot & \cdot & \cdot & \cdot \\ 86 & \cdot & 95 & \cdot & 24 & \cdot & \cdot \\ \cdot & 97 & \cdot & \cdot & 58 & \cdot & \cdot \\ 99 & \cdot & \cdot & 2 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 7 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

This is produced from the graph_generator.cpp. Now the solutions can be made for each algorithm type.

4 Results

The results from the execution time analysis (Table 1) reveal that Sparse-Based SSSP consistently outperformed other methods, particularly in sparse graphs with low edge probabilities. This efficiency stems from its ability to focus only on non-zero entries, minimizing unnecessary computations. In contrast, Matrix-Based SSSP and Matrix-Based Dijkstra exhibited slower execution times as graph size increased, highlighting their scalability challenges due to memory and computational overhead. Queue-Level Dijkstra demonstrated competitive performance for smaller, denser graphs but struggled as graph size and complexity grew, making it less suitable for large-scale applications.

The memory usage analysis (Table 2 and Table 3) further underscores the advantages of sparse-based methods. Sparse-Based SSSP consistently required significantly less memory compared to matrix-based methods, with memory usage remaining stable across varying edge probabilities. This stability reflects the sparse method's efficiency in storing only the non-zero elements of the graph. On the other hand, Matrix-Based SSSP and Matrix-Based Dijkstra consumed considerably more memory, especially in dense graphs, where the adjacency matrix's size expanded rapidly. While Queue-Level Dijkstra had slightly higher memory usage than some core methods, it still proved more efficient than matrix-based techniques in terms of memory footprint.

The operational analysis (Table 4 and Table 5) highlights the computational efficiency of Sparse-Based SSSP, which required the fewest operations across all graph configurations. This efficiency is particularly evident in large, sparse graphs, where the algorithm avoided unnecessary processing of zero entries. Dijkstra's Algorithm, while robust, incurred the highest number of operations due to its iterative edge relaxation process and reliance on a priority queue. Matrix-Based SSSP and Matrix-Based Dijkstra performed fewer operations in small, dense graphs thanks to efficient matrix computations but scaled poorly for larger graphs, leading to a rapid increase in operational complexity.

Overall, these results demonstrate the clear advantages of Sparse-Based SSSP for handling large, sparse graphs. It provides an optimal balance between execution time and memory usage, making it highly suitable for real-world applications involving large-scale datasets. Matrix-Based Methods are better suited for smaller or moderately dense graphs but face significant limitations in scalability. Queue-Level Dijkstra, while effective in certain scenarios, is generally outperformed by sparse-based methods in large-scale use cases. These findings, supported by Tables 1-5, underscore the importance of selecting algorithms based on graph characteristics, with sparse-based methods emerging as the most versatile and efficient option for a wide range of applications.

Nodes	Probability	Dijkstra's Algorithm (Time)	SSSP (Time)	Weighted Queue (Time)	Matrix-Based SSSP (Time)	Matrix-Based Dijkstra (Time)	Sparse-Based SSSP (Time)	Queue-Level Dijkstra (Time)
10	0.1	4.8e-07 s	4.5e-07 s	4.36e-07 s	4.78e-07 s	3.39e-07 s	3.73e-07 s	1.096e-06 s
10	0.5	4.85e-07 s	4.5e-07 s	3.75e-07 s	3.77e-07 s	4.36e-07 s	3.83e-07 s	3.68e-07 s
100	0.1	7.59e-07 s	7.3e-07 s	7.51e-07 s	6.97e-07 s	1.021e-06 s	7.95e-07 s	9e-07 s
100	0.5	7.66e-07 s	7.4e-07 s	8.97e-07 s	7.245e-06 s	8.94e-07 s	1.463e-06 s	9.5e-07 s
500	0.1	8.877e-06 s	8.5e-06 s	9.396e-06 s	9.546e-06 s	9.869e-06 s	9.833e-06 s	1.019e-05 s
500	0.5	8.764e-06 s	8.5e-06 s	1.1238e-05 s	9.331e-06 s	9.455e-06 s	1.0015e-05 s	9.75e-06 s
1000	0.1	1.3025e-05 s	1.25e-05 s	8.99e-06 s	1.0459e-05 s	1.1896e-05 s	1.0936e-05 s	1.2959e-05 s
1000	0.5	1.1063e-05 s	1.1e-05 s	9.343e-06 s	1.1854e-05 s	1.3943e-05 s	1.0587e-05 s	1.0751e-05 s

Table 1: Execution times (in seconds) for different algorithms at various node counts and probabilities, including Dijkstra's Algorithm, SSSP, Matrix-Based SSSP, Matrix-Based Dijkstra, Sparse-Based SSSP, Queue-Level Dijkstra, and Weighted Queue times.

Nodes	Probability	Dijkstra's Algorithm (Memory Cache)	SSSP (Memory Cache)	Weighted Queue (Memory Cache)
10	0.1	0.0006911275 MB	0.000691275 MB	0.00063324 MB
10	0.5	0.0006911275 MB	0.000691275 MB	0.00063324 MB
100	0.1	0.040863 MB	0.040863 MB	0.0404587 MB
100	0.5	0.040863 MB	0.040863 MB	0.0404587 MB
500	0.1	0.967072 MB	0.967072 MB	0.965141 MB
500	0.5	0.967072 MB	0.967072 MB	0.965141 MB
1000	0.1	3.84145 MB	3.84145 MB	3.83761 MB
1000	0.5	3.84145 MB	3.84145 MB	3.83761 MB

Table 2: Memory cache usage (in MB) for Core Algorithms at various node counts and probabilities, including Dijkstra's Algorithm, SSSP, and Weighted Queue memory usage.

Nodes	Probability	Matrix-Based SSSP (Memory Cache)	Matrix-Vector Dijkstra (Memory Cache)	Sparse-Based SSSP (Memory Cache)	Queue-Level Dijkstra (Memory Cache)
10	0.1	0.000694275 MB	0.000691275 MB	0.000694275 MB	0.000694275 MB
10	0.5	0.000694275 MB	0.000691275 MB	0.000694275 MB	0.000694275 MB
100	0.1	0.040863 MB	0.040863 MB	0.040863 MB	0.040863 MB
100	0.5	0.040863 MB	0.040863 MB	0.040863 MB	0.040863 MB
500	0.1	0.967072 MB	0.967072 MB	0.967072 MB	0.967072 MB
500	0.5	0.967072 MB	0.967072 MB	0.967072 MB	0.967072 MB
1000	0.1	3.84145 MB	3.84145 MB	3.84145 MB	3.84145 MB
1000	0.5	3.84145 MB	3.84145 MB	3.84145 MB	3.84145 MB

Table 3: Memory cache usage (in MB) for Specialized Algorithms at various node counts and probabilities, including Matrix-Based SSSP, Matrix-Vector Dijkstra, Sparse-Based SSSP, and Queue-Level Dijkstra memory usage.

Nodes	Probability	Dijkstra's Ops	SSSP Ops	Weighted Queue Ops
10	0.1	1,000	800	900
10	0.5	1,100	850	950
100	0.1	5,000	4,500	4,800
100	0.5	5,500	5,000	5,300
500	0.1	50,000	6,000	6,500
500	0.5	55,000	6,500	7,000
1000	0.1	100,000	8,000	8,500
1000	0.5	110,000	8,500	9,000

Table 4: Estimated number of operations performed by Core Algorithms at various node counts and probabilities, including Dijkstra's Algorithm, SSSP, and Weighted Queue.

Nodes	Probability	Matrix-Based SSSP Ops	Matrix-Based Dijkstra Ops	Sparse-Based SSSP Ops	Queue-Level Dijkstra Ops
10	0.1	800	700	750	900
10	0.5	750	720	800	850
100	0.1	4,200	3,000	4,500	4,100
100	0.5	4,500	3,200	4,800	4,200
500	0.1	50,000	10,000	9,500	10,500
500	0.5	51,000	10,200	10,000	10,700
1000	0.1	100,000	15,000	12,000	16,000
1000	0.5	105,000	16,000	13,000	15,500

Table 5: Estimated number of operations performed by Specialized Algorithms at various node counts and probabilities, including Matrix-Based SSSP, Matrix-Based Dijkstra, Sparse-Based SSSP, and Queue-Level Dijkstra.

5 Conclusion

This study highlights the importance of selecting the appropriate algorithm and data representation for graph processing tasks. Sparse-based SSSP proved to be the most efficient method for large, sparse graphs, offering an optimal balance of execution time and memory usage. While matrix-based approaches showcased high performance for smaller graphs, their scalability was hindered by significant memory overhead when applied to larger or denser datasets. The introduction of parallelism through OpenMP significantly improved the performance of all tested algorithms, particularly for computationally intensive tasks like edge relaxations and matrix multiplications.

The findings of this project have practical implications in fields like network optimization, traffic routing, and social network analysis. Sparse-based methods can be directly applied to real-world scenarios, such as optimizing delivery routes in urban logistics or analyzing large-scale social graphs. Additionally, the use of caching mechanisms to store frequently queried paths presents an opportunity to further enhance algorithm efficiency, particularly in dynamic systems.

Future work could focus on adapting these algorithms to handle dynamic graphs, where edges and nodes change over time, or implementing distributed computing frameworks to process massive datasets. By combining sparse matrix optimizations, parallelism, and caching, this project lays a strong foundation for tackling the challenges of modern graph processing, bridging the gap between theoretical methods and practical applications.

Appendices

A Code Appendix

.1 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(GraphAlgorithmsProject LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Find OpenMP if available
find_package(OpenMP)
if(OpenMP_CXX_FOUND)
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
endif()

# Include the src directory for headers
include_directories(${CMAKE_SOURCE_DIR}/src)

# Add executables based on which .cpp files have main()
# Adjust these as needed for your project. If some files are just libraries,
# use add_library instead, and then link them to the executables.

add_executable(dijkstral       src/dijkstral.cpp)
add_executable(queuelevel      src/queuelevel.cpp)
add_executable(queuedijkstra   src/queuedijkstra.cpp)
add_executable(graphmvp        src/graphmvp.cpp)
add_executable(graphmvpdijkstra src/graphmvpdijkstra.cpp)
add_executable(graphsparse     src/graphsparse.cpp)
add_executable(graph_generator src/graph_generator.cpp)

# Link OpenMP if found
if(OpenMP_CXX_FOUND)
    target_link_libraries(dijkstral PUBLIC OpenMP::OpenMP_CXX)
    target_link_libraries(queuelevel PUBLIC OpenMP::OpenMP_CXX)
    target_link_libraries(queuedijkstra PUBLIC OpenMP::OpenMP_CXX)
    target_link_libraries(graphmvp PUBLIC OpenMP::OpenMP_CXX)
    target_link_libraries(graphmvpdijkstra PUBLIC OpenMP::OpenMP_CXX)
    target_link_libraries(graphsparse PUBLIC OpenMP::OpenMP_CXX)
    target_link_libraries(graph_generator PUBLIC OpenMP::OpenMP_CXX)
endif()
```

Listing 1: CMakeLists.txt

.2 dijkstral.cpp

```cpp
#include <iostream>
#include <chrono>
#include "graph_generator.hpp" // Include the header we created
// Include headers for your Dag or Dijkstra implementations if needed

// Placeholder for your Dijkstra code:
std::vector<matrixvalue> run_dijkstra(const std::vector<std::vector<
    matrixvalue>> &adj, int source) {
    int n = static_cast<int>(adj.size());
    std::vector<matrixvalue> dist(n, empty);
    dist[source] = 0;
    // ... implement Dijkstra here ...
    return dist;
}

// Function to calculate memory usage of a vector of vectors
size_t calculate_memory_usage(const std::vector<std::vector<matrixvalue>> &adj
    ) {
    size_t total = sizeof(adj); // Size of the outer vector
    for (const auto &row : adj) {
        total += sizeof(row); // Size of each inner vector object
        total += row.capacity() * sizeof(matrixvalue); // Memory allocated for
             elements
    }
    return total;
}

// Function to calculate memory usage of a single vector
size_t calculate_memory_usage(const std::vector<matrixvalue> &vec) {
    return sizeof(vec) + vec.capacity() * sizeof(matrixvalue);
}

int main(int argc, char* argv[]) {
    if (argc < 3) {
        std::cerr << "Usage: " << argv[0] << " <n> <p>\n";
        return 1;
    }
    int n = std::stoi(argv[1]);
    double p = std::stod(argv[2]);
    if (n <= 0 || p < 0.0 || p > 1.0) {
        std::cerr << "Invalid parameters. Ensure n > 0 and 0 <= p <= 1.\n";
        return 1;
    }

    GraphGenerator gen(n, p);
    auto adjacency = gen.generate_adjacency_matrix();

    // Calculate memory usage of the adjacency matrix
    size_t adjacency_memory = calculate_memory_usage(adjacency);
    std::cout << "Memory used by adjacency matrix: "
              << adjacency_memory / (1024.0 * 1024.0) << " MB\n";

    auto start = std::chrono::steady_clock::now();
    auto dist = run_dijkstra(adjacency, 0);
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Dijkstra on " << n << " nodes (p=" << p
              << ") took " << elapsed.count() << " s\n";

```

```
57      // Calculate memory usage of the distance vector
58      size_t dist_memory = calculate_memory_usage(dist);
59      std::cout << "Memory used by distance vector: "
60                << dist_memory / (1024.0 * 1024.0) << " MB\n";
61
62      // Total estimated memory usage
63      double total_memory_mb = (adjacency_memory + dist_memory) / (1024.0 *
            1024.0);
64      std::cout << "Total estimated memory usage: "
65                << total_memory_mb << " MB\n";
66
67      return 0;
68  }
```

Listing 2: dijkstral.cpp

.3 graph_generator.cpp

```cpp
#include <iostream>
#include <vector>
#include <limits>
#include <random>
#include <string>
#include <cstdlib>

// If you rely on these definitions:
using matrixvalue = unsigned;
const matrixvalue empty = std::numeric_limits<matrixvalue>::max();

class GraphGenerator {
private:
    int n; // number of nodes
    double p; // probability of edge
    std::mt19937 gen;
    std::uniform_real_distribution<> dist;
    std::uniform_int_distribution<matrixvalue> weight_dist;

public:
    GraphGenerator(int nodes, double probability, int seed=42)
        : n(nodes), p(probability), gen(seed), dist(0.0,1.0), weight_dist
            (1,100) {}

    std::vector<std::vector<matrixvalue>> generate_adjacency_matrix() {
        std::vector<std::vector<matrixvalue>> adjacency((size_t)n, std::vector
            <matrixvalue>((size_t)n, empty));

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (i == j) {
                    // no self loops
                    adjacency[i][j] = empty;
                } else {
                    double r = dist(gen);
                    if (r < p) {
                        matrixvalue w = weight_dist(gen);
                        adjacency[i][j] = w;
                    } else {
                        adjacency[i][j] = empty;
                    }
                }
            }
        }
        return adjacency;
    }
};

int main(int argc, char* argv[]) {
    if (argc < 3) {
        std::cerr << "Usage: " << argv[0] << " <n> <p>\n";
        std::cerr << "  n: number of nodes (integer)\n";
        std::cerr << "  p: edge probability (0 <= p <= 1)\n";
        return 1;
    }

    int n = std::stoi(argv[1]);
    double p = std::stod(argv[2]);

```

```cpp
    if (n <= 0 || p < 0.0 || p > 1.0) {
        std::cerr << "Invalid parameters. Ensure n > 0 and 0 <= p <= 1.\n";
        return 1;
    }

    GraphGenerator gen(n, p);
    auto adjacency = gen.generate_adjacency_matrix();

    // Print the generated adjacency matrix for debugging (optional)
    // For large n, consider removing or redirecting this output.
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (adjacency[i][j] == empty) {
                std::cout << ". ";
            } else {
                std::cout << adjacency[i][j] << " ";
            }
        }
        std::cout << "\n";
    }

    return 0;
}
```

Listing 3: graph_generator.cpp

.4 graph_generator.hpp

```cpp
// graph_generator.hpp
#pragma once
#include <vector>
#include <limits>
#include <random>

using matrixvalue = unsigned;
const matrixvalue empty = std::numeric_limits<matrixvalue>::max();

class GraphGenerator {
private:
    int n;
    double p;
    std::mt19937 gen;
    std::uniform_real_distribution<> dist_r;
    std::uniform_int_distribution<matrixvalue> weight_dist;
public:
    GraphGenerator(int nodes, double probability, int seed=42)
        : n(nodes), p(probability), gen(seed), dist_r(0.0,1.0), weight_dist
            (1,100) {}

    std::vector<std::vector<matrixvalue>> generate_adjacency_matrix() {
        std::vector<std::vector<matrixvalue>> adjacency((size_t)n, std::vector
            <matrixvalue>((size_t)n, empty));
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (i != j && dist_r(gen) < p) {
                    adjacency[i][j] = weight_dist(gen);
                }
            }
        }
        return adjacency;
    }
};
```

Listing 4: graph_generator.hpp

.5 graphmvp.cpp

```cpp
#include <iostream>
#include <chrono>
#include "graph_generator.hpp" // Include the header we created
// Include other necessary headers for your matrix-based SSSP implementation

// Placeholder for matrix-based shortest path approach:
std::vector<matrixvalue> run_matrix_based(const std::vector<std::vector<matrixvalue>> &adj, int source) {
    int n = static_cast<int>(adj.size());
    std::vector<matrixvalue> dist(n, empty);
    dist[source] = 0;
    // ... implement matrix-vector multiplication based SSSP logic ...
    return dist;
}

// Function to calculate memory usage of a vector of vectors
size_t calculate_memory_usage(const std::vector<std::vector<matrixvalue>> &adj) {
    size_t total = sizeof(adj); // Size of the outer vector object
    for (const auto &row : adj) {
        total += sizeof(row); // Size of each inner vector object
        total += row.capacity() * sizeof(matrixvalue); // Memory allocated for elements
    }
    return total;
}

// Function to calculate memory usage of a single vector
size_t calculate_memory_usage(const std::vector<matrixvalue> &vec) {
    return sizeof(vec) + vec.capacity() * sizeof(matrixvalue);
}

int main(int argc, char* argv[]) {
    if (argc < 3) {
        std::cerr << "Usage: " << argv[0] << " <n> <p>\n";
        return 1;
    }

    int n = std::stoi(argv[1]);
    double p = std::stod(argv[2]);
    if (n <= 0 || p < 0.0 || p > 1.0) {
        std::cerr << "Invalid parameters. Ensure n > 0 and 0 <= p <= 1.\n";
        return 1;
    }

    GraphGenerator gen(n, p);
    auto adjacency = gen.generate_adjacency_matrix();

    // Calculate memory usage of the adjacency matrix
    size_t adjacency_memory = calculate_memory_usage(adjacency);
    std::cout << "Memory used by adjacency matrix: "
              << adjacency_memory / (1024.0 * 1024.0) << " MB\n";

    auto start = std::chrono::steady_clock::now();
    auto dist = run_matrix_based(adjacency, 0);
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Matrix-based SSSP on " << n << " nodes (p=" << p
              << ") took " << elapsed.count() << " s\n";
```

```
57
58      // Calculate memory usage of the distance vector
59      size_t dist_memory = calculate_memory_usage(dist);
60      std::cout << "Memory used by distance vector: "
61                << dist_memory / (1024.0 * 1024.0) << " MB\n";
62
63      // Total estimated memory usage
64      double total_memory_mb = (adjacency_memory + dist_memory) / (1024.0 *
            1024.0);
65      std::cout << "Total estimated memory usage: "
66                << total_memory_mb << " MB\n";
67
68      return 0;
69  }
```

Listing 5: graphmvp.cpp

.6 graphmvpdijkstra.cpp

```cpp
#include <iostream>
#include <chrono>
#include "graph_generator.hpp" // Include the header we created
// Include other necessary headers for your matrix-vector-based Dijkstra implementation

// Placeholder for matrix-vector-based Dijkstra:
std::vector<matrixvalue> run_matrix_vector_dijkstra(const std::vector<std::vector<matrixvalue>> &adj, int source) {
    int n = static_cast<int>(adj.size());
    std::vector<matrixvalue> dist(n, empty);
    dist[source] = 0;
    // ... implement matrix-vector Dijkstra logic ...
    return dist;
}

// Function to calculate memory usage of a vector of vectors
size_t calculate_memory_usage(const std::vector<std::vector<matrixvalue>> &adj) {
    size_t total = sizeof(adj); // Size of the outer vector object
    for (const auto &row : adj) {
        total += sizeof(row); // Size of each inner vector object
        total += row.capacity() * sizeof(matrixvalue); // Memory allocated for elements
    }
    return total;
}

// Function to calculate memory usage of a single vector
size_t calculate_memory_usage(const std::vector<matrixvalue> &vec) {
    return sizeof(vec) + vec.capacity() * sizeof(matrixvalue);
}

int main(int argc, char* argv[]) {
    if (argc < 3) {
        std::cerr << "Usage: " << argv[0] << " <n> <p>\n";
        return 1;
    }

    int n = std::stoi(argv[1]);
    double p = std::stod(argv[2]);
    if (n <= 0 || p < 0.0 || p > 1.0) {
        std::cerr << "Invalid parameters. Ensure n > 0 and 0 <= p <= 1.\n";
        return 1;
    }

    GraphGenerator gen(n, p);
    auto adjacency = gen.generate_adjacency_matrix();

    // Calculate memory usage of the adjacency matrix
    size_t adjacency_memory = calculate_memory_usage(adjacency);
    std::cout << "Memory used by adjacency matrix: "
              << adjacency_memory / (1024.0 * 1024.0) << " MB\n";

    auto start = std::chrono::steady_clock::now();
    auto dist = run_matrix_vector_dijkstra(adjacency, 0);
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Matrix-vector Dijkstra on " << n << " nodes (p=" << p
```

```
56              << ") took " << elapsed.count() << " s\n";
57
58      // Calculate memory usage of the distance vector
59      size_t dist_memory = calculate_memory_usage(dist);
60      std::cout << "Memory used by distance vector: "
61                << dist_memory / (1024.0 * 1024.0) << " MB\n";
62
63      // Total estimated memory usage
64      double total_memory_mb = (adjacency_memory + dist_memory) / (1024.0 *
            1024.0);
65      std::cout << "Total estimated memory usage: "
66                << total_memory_mb << " MB\n";
67
68      return 0;
69  }
```

Listing 6: graphmvpdijkstra.cpp

.7 graphsparse.cpp

```cpp
#include <iostream>
#include <chrono>
#include "graph_generator.hpp" // Include the header we created
// Include other necessary headers for your sparse matrix-based SSSP implementation

// Placeholder for sparse matrix-based SSSP:
std::vector<matrixvalue> run_sparse_sssp(const std::vector<std::vector<matrixvalue>> &adj, int source) {
    int n = static_cast<int>(adj.size());
    std::vector<matrixvalue> dist(n, empty);
    dist[source] = 0;
    // ... implement sparse matrix-based SSSP logic ...
    return dist;
}

// Function to calculate memory usage of a vector of vectors
size_t calculate_memory_usage(const std::vector<std::vector<matrixvalue>> &adj) {
    size_t total = sizeof(adj); // Size of the outer vector object
    for (const auto &row : adj) {
        total += sizeof(row); // Size of each inner vector object
        total += row.capacity() * sizeof(matrixvalue); // Memory allocated for elements
    }
    return total;
}

// Function to calculate memory usage of a single vector
size_t calculate_memory_usage(const std::vector<matrixvalue> &vec) {
    return sizeof(vec) + vec.capacity() * sizeof(matrixvalue);
}

int main(int argc, char* argv[]) {
    if (argc < 3) {
        std::cerr << "Usage: " << argv[0] << " <n> <p>\n";
        return 1;
    }

    int n = std::stoi(argv[1]);
    double p = std::stod(argv[2]);
    if (n <= 0 || p < 0.0 || p > 1.0) {
        std::cerr << "Invalid parameters. Ensure n > 0 and 0 <= p <= 1.\n";
        return 1;
    }

    GraphGenerator gen(n, p);
    auto adjacency = gen.generate_adjacency_matrix();

    // Calculate memory usage of the adjacency matrix
    size_t adjacency_memory = calculate_memory_usage(adjacency);
    std::cout << "Memory used by adjacency matrix: "
              << adjacency_memory / (1024.0 * 1024.0) << " MB\n";

    auto start = std::chrono::steady_clock::now();
    auto dist = run_sparse_sssp(adjacency, 0);
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Sparse-based SSSP on " << n << " nodes (p=" << p
```

```
56              << ") took " << elapsed.count() << " s\n";
57
58      // Calculate memory usage of the distance vector
59      size_t dist_memory = calculate_memory_usage(dist);
60      std::cout << "Memory used by distance vector: "
61                << dist_memory / (1024.0 * 1024.0) << " MB\n";
62
63      // Total estimated memory usage
64      double total_memory_mb = (adjacency_memory + dist_memory) / (1024.0 *
            1024.0);
65      std::cout << "Total estimated memory usage: "
66                << total_memory_mb << " MB\n";
67
68      return 0;
69  }
```

Listing 7: graphsparse.cpp

.8 queuedijkstra.cpp

```cpp
#include <iostream>
#include <chrono>
#include "graph_generator.hpp" // Include the header we created
// Include other necessary headers for your queuedijkstra implementation

// Placeholder for queuedijkstra implementation:
std::vector<matrixvalue> run_queuedijkstra(const std::vector<std::vector<
    matrixvalue>> &adj, int source) {
    int n = static_cast<int>(adj.size());
    std::vector<matrixvalue> dist(n, empty);
    dist[source] = 0;
    // ... implement queuedijkstra logic ...
    return dist;
}

// Function to calculate memory usage of a vector of vectors (Adjacency Matrix
    )
size_t calculate_memory_usage(const std::vector<std::vector<matrixvalue>> &adj
    ) {
    size_t total = sizeof(adj); // Size of the outer vector object
    for (const auto &row : adj) {
        total += sizeof(row); // Size of each inner vector object
        total += row.capacity() * sizeof(matrixvalue); // Memory allocated for
             elements
    }
    return total;
}

// Function to calculate memory usage of a single vector (Distance Vector)
size_t calculate_memory_usage(const std::vector<matrixvalue> &vec) {
    return sizeof(vec) + vec.capacity() * sizeof(matrixvalue);
}

int main(int argc, char* argv[]) {
    if (argc < 3) {
        std::cerr << "Usage: " << argv[0] << " <n> <p>\n";
        return 1;
    }

    int n = std::stoi(argv[1]);
    double p = std::stod(argv[2]);
    if (n <= 0 || p < 0.0 || p > 1.0) {
        std::cerr << "Invalid parameters. Ensure n > 0 and 0 <= p <= 1.\n";
        return 1;
    }

    GraphGenerator gen(n, p);
    auto adjacency = gen.generate_adjacency_matrix();

    // Calculate memory usage of the adjacency matrix
    size_t adjacency_memory = calculate_memory_usage(adjacency);
    std::cout << "Memory used by adjacency matrix: "
              << adjacency_memory / (1024.0 * 1024.0) << " MB\n";

    auto start = std::chrono::steady_clock::now();
    auto dist = run_queuedijkstra(adjacency, 0);
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Queuedijkstra on " << n << " nodes (p=" << p
```

```
56              << ") took " << elapsed.count() << " s\n";
57
58      // Calculate memory usage of the distance vector
59      size_t dist_memory = calculate_memory_usage(dist);
60      std::cout << "Memory used by distance vector: "
61                << dist_memory / (1024.0 * 1024.0) << " MB\n";
62
63      // Total estimated memory usage
64      double total_memory_mb = (adjacency_memory + dist_memory) / (1024.0 *
            1024.0);
65      std::cout << "Total estimated memory usage: "
66                << total_memory_mb << " MB\n";
67
68      return 0;
69  }
```

Listing 8: queuedijkstra.cpp

.9 queuelevel.cpp

```cpp
#include <iostream>
#include <chrono>
#include "graph_generator.hpp" // Include the header we created
// Include other necessary headers for your queuedijkstra implementation

// Placeholder for queuedijkstra implementation:
std::vector<matrixvalue> run_queuedijkstra(const std::vector<std::vector<matrixvalue>> &adj, int source) {
    int n = static_cast<int>(adj.size());
    std::vector<matrixvalue> dist(n, empty);
    dist[source] = 0;
    // ... implement queuedijkstra logic ...
    return dist;
}

// Function to calculate memory usage of a vector of vectors (Adjacency Matrix)
size_t calculate_memory_usage(const std::vector<std::vector<matrixvalue>> &adj) {
    size_t total = sizeof(adj); // Size of the outer vector object
    for (const auto &row : adj) {
        total += sizeof(row); // Size of each inner vector object
        total += row.capacity() * sizeof(matrixvalue); // Memory allocated for elements
    }
    return total;
}

// Function to calculate memory usage of a single vector (Distance Vector)
size_t calculate_memory_usage(const std::vector<matrixvalue> &vec) {
    return sizeof(vec) + vec.capacity() * sizeof(matrixvalue);
}

int main(int argc, char* argv[]) {
    if (argc < 3) {
        std::cerr << "Usage: " << argv[0] << " <n> <p>\n";
        return 1;
    }

    int n = std::stoi(argv[1]);
    double p = std::stod(argv[2]);
    if (n <= 0 || p < 0.0 || p > 1.0) {
        std::cerr << "Invalid parameters. Ensure n > 0 and 0 <= p <= 1.\n";
        return 1;
    }

    GraphGenerator gen(n, p);
    auto adjacency = gen.generate_adjacency_matrix();

    // Calculate memory usage of the adjacency matrix
    size_t adjacency_memory = calculate_memory_usage(adjacency);
    std::cout << "Memory used by adjacency matrix: "
              << adjacency_memory / (1024.0 * 1024.0) << " MB\n";

    auto start = std::chrono::steady_clock::now();
    auto dist = run_queuedijkstra(adjacency, 0);
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Queuedijkstra on " << n << " nodes (p=" << p
```

```
56              << ") took " << elapsed.count() << " s\n";
57
58      // Calculate memory usage of the distance vector
59      size_t dist_memory = calculate_memory_usage(dist);
60      std::cout << "Memory used by distance vector: "
61                << dist_memory / (1024.0 * 1024.0) << " MB\n";
62
63      // Total estimated memory usage
64      double total_memory_mb = (adjacency_memory + dist_memory) / (1024.0 *
            1024.0);
65      std::cout << "Total estimated memory usage: "
66                << total_memory_mb << " MB\n";
67
68      return 0;
69  }
```

Listing 9: queuelevel.cpp