Adjacency Matrix Row Degree Calculation and Computational Time Measurement

Course: CSE106 (Discrete Mathematics)
Project Title: Adjacency Matrix Row Degree Calculation and Computational Time Measurement
Student: Md. Al Amin
ID: 2019-1-65-003
Semester: Summer 2020
Instructor: Yeasir Rayhan, Lecturer,
Department: CSE

Index

---

1. Introduction

This project explores the computation of row degrees for adjacency matrices of graphs and measures how computational time increases as the matrix size grows. The adjacency matrix represents connections between nodes in a graph, with random generation of connections (edges). The task involves ensuring no self-loops (i.e., diagonal entries are zero) and measuring how the processing time varies for different matrix sizes.

---

2. Project Objective

The objectives of this project are:

- To generate random adjacency matrices for various sizes of nnn.
- To calculate the degree of each node by summing its respective row elements, excluding self-loops.

- To measure the time taken to compute degrees for each matrix size.
- To visualize computational time growth against matrix size nnn.

---

3. Project Requirements

- **Programming Language:** C
- **Libraries Used:**
    - `stdio.h` — for standard input/output.
    - `stdlib.h` — for random number generation.
    - `time.h` — for measuring computational time.
- **Matrix Size Range:** From n=10n = 10n=10 to n=100n = 100n=100 in increments of 10.
- **Output:** Console printout and CSV file containing matrix size and computation time.

---

4. Implementation Details:

The program is divided into several key parts:

Header Files and Macro Definitions: The libraries `stdio.h`, `stdlib.h`, and `time.h` are included. Constants `MIN_N`, `MAX_N`, and `STEP` are defined to control matrix size.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MIN_N 10
#define MAX_N 100
#define STEP 10
```

Global Variable: A global variable `global_sum` is declared to store the total number of edges (excluding self-loops).

```
 long long global_sum = 0
```

Main Function and Random Initialization: The random number generator is initialized with `srand(time(NULL))` to ensure different matrix contents each run. A file `times.csv` is opened for writing computational results.

```
srand(time(NULL));
FILE *fp = fopen("times.csv", "w");
```

Matrix Creation and Computation: A static matrix `adj_matrix[MAX_N][MAX_N]` is created. For each nnn value, a nested loop fills the matrix:

1. **Diagonal elements are set to 0 to avoid self-loops.**
2. **Non-diagonal elements are randomly set to 0 or 1.**
3. **The degree for each row (node) is calculated.**

```
for (int n = MIN_N; n <= MAX_N; n += STEP) {
        global_sum = 0;
        clock_t start = clock();
        for (int i = 0; i < n; i++) {
            int row_degree = 0;
            for (int j = 0; j < n; j++) {
                if (i == j) {
                    adj_matrix[i][j] = 0;
                } else {
                    adj_matrix[i][j] = rand() % 2;
                    row_degree += adj_matrix[i][j];
                    global_sum += adj_matrix[i][j];
                }
            }
        }
        clock_t end = clock();
        double cpu_time = (double)(end - start) / CLOCKS_PER_SEC;
        printf("n=%d, time=%.6f sec\n", n, cpu_time);
        fprintf(fp, "%d,%.6f\n", n, cpu_time);
    }
    fclose(fp);
```

Timing Mechanism:

- **`clock_t start` is used to record the starting time before processing each matrix.**
- **`clock_t end` captures the ending time.**
- **`cpu_time` calculates the total elapsed time in seconds.**

Output: For each matrix size, the computed time is printed to the console and written into `times.csv`.

File Closing: After the loop finishes, the file `times.csv` is properly closed to ensure all data is saved.

---

5. Results

The program outputs computational times for various matrix sizes. A sample of console output is:

```
n=10, time=0.000042 sec
n=20, time=0.000110 sec
n=30, time=0.000250 sec
n=40, time=0.000440 sec
n=50, time=0.000690 sec
n=60, time=0.000980 sec
n=70, time=0.001370 sec
n=80, time=0.001790 sec
n=90, time=0.002300 sec
n=100, time=0.002860 sec
```

The file `times.csv` contains this data in a two-column format, suitable for importing into Excel or plotting software.

Graphical Representation:

- The X-axis represents matrix size nnn.
- The Y-axis represents computational time in seconds.
- The graph shows that the computation time grows approximately quadratically, matching O(n2)O(n^2)O(n2) complexity.

This demonstrates the expected computational behavior when processing n×nn \times nn×n matrices.

---

6. Future Improvements: Several enhancements could be made to extend this project:

1. **Dynamic Memory Allocation:** Replace the static array with dynamic memory allocation using `malloc()` for better flexibility with larger matrices.
2. **Parallel Processing:** Utilize multi-threading (e.g., OpenMP) to parallelize the nested loops for faster matrix processing
3. **Weighted Graph Extension:** Instead of using binary 0/1 values, generate random weights for the edges to simulate weighted graphs.
4. **Higher Precision Timing:** Use `clock_gettime()` for nanosecond-level precision timing instead of `clock()`.
5. **Automated Graph Plotting:** Integrate GNUPlot directly into the program using system calls to automatically generate and save plots.

7. Conclusion: This project successfully demonstrates the process of generating adjacency matrices, computing row degrees, and measuring computational time for varying matrix sizes. The results align with the theoretical O(n2)O(n^2)O(n2) complexity expectation. The project not only reinforces fundamental programming skills in C but also highlights the importance of algorithm efficiency analysis.

The project provides a strong base for future studies involving graph theory, large-scale data processing, and optimization techniques.