# HPGDA Data Structure Report

Alessio Braccini

June 2022

## 1 Introduction

This work is part of a contest organized by Oracle. The challenge consists in implementing a graph data structure to store a graph that is efficient both in terms of memory usage and run-time performance and algorithms that efficiently populate this data structure with real graph data and compute some searching like Breadth First Search and Depth First Search.

## 2 Data Description

The provided datasests are taken from the Linked Data Benchmark Council (LDBC). These datasets are composed by two files: a *.v* file containing the list of the vertices of the graph and a *.e* one containing the list of the edges.

In advance the *.e* file is composed by three elements: source vertex, destination vertex and weight (if not present the program set it to 1).

## 3 Data Preparation

### 3.1 Program launching and preparation

To run the program some arguments are needed:

- *graph name* is a required parameter that tells the position of the graph in memory;
- *source vertex* is a required parameter that tells the vertex where the BFS and DFS algorithms has to start the search;
- **-U** is an optional parameter that tells if a graph is undirected, if omitted the graph will be processed as a directed one;
- **-d** is an optional parameter that tells if the program will run in debug mode or not (the debug mode adds some useful print to better understand the graph characteristics).

### 3.2 Data loading in main memory

The second stage starts with the reading of the data from the file. Then they are loaded in a temporary data structure. The code that makes this procedure is already provided and it can be found in the utils.h header. The pre implemented data structure is an array of tuple for the edges and a set for the nodes.

I changed this structure and switch to a vector because for my implementation it was needed an ordinate sequence of edges, ordered by source vertex. The object vector , implemented in c++, provides a sort method. By paying few time in ordering (the pre implemented method has time complexity of $O(n)$).

Some changes in utils.h and in GraphAlgorithm.h headers were made in order to adapt this new data structure.

It was added a new header class called triple.h that describes the element that the temporary vector has to store. More precisely, triple is an object that contains 3 elements:

- **x**: that store the source vertex;
- **y**: that store the destination vertex;
- **val**: that store the weight of the vertex.

## 4 Data structure

The chosen representation to store the graph is the Compressed Sparse Row (CSR). This representation has been selected because of the really fast operations speed. I personally preferred CSR over COO representation because the high gain in speed computation when operations are performed, like get_neighbors method that is performed many times.

This representation is build with three arrays:

- **col_idx**:
- **row_ptr**:
- **value**:

It doesn't need a constructor because these arrays are declared as vector object, so the implementation is able to manage the construction and the realloc if the structure ran out of size.

### 4.1 Motivations of CSR

Recalling what said before the main reason in the CSR choice is the fast speed in operation. Another motivation is the memory impact because arrays don't need other accessories structure i.e. pointers like in lists, etc...

### 4.2 Population

The population is performed in $O(n)$ where $n$ is the number of element of the *.e* file. It is composed by two *for-loop*:

- **Main for-loop**: it scans every edge and fills the three CSR data structures
- **Secondary for-loop**: at the end of the main loop it completes the row_ptr vector. This is required because if some last rows of the CSR matrix have no elements they are not considered in the structure and this can lead to some error when get_neighbors is execute.

This method need a sorted input because it simplifies a lot the procedure and also speeds up the process. The row_ptr vector needs sorted input because it keeps the count of the previous row, if an unsorted input is used the sorting phase has to be done at this stage and the complexity increases a lot.

### 4.3 Get Neighbors

The get neighbors is performed in $O(e)$ where $e$ is the number of the NNZ elements in the row of the CSR matrix. The method exploits a characteristic of the CSR: the neighbors of a vertex are the NNZ elements in the row of the CSR matrix. This can be done in two steps:

- **Computation of Number of NNZ element**: by exploiting the sorting of row_ptr we can easily calculate the number of NNZ element that are in the CSR matrix row. This is done (in pseudo-code) by:

  ```
  num_elem = row_ptr[vertex] -row_ptr[vertex-1]
  ```

- **Creation of the returned object**: when the number of the elements is known we then create an object that will be filled and returned. The chosen object is a vector$\langle$pair$\langle long, double\rangle\rangle$ that contains the neighbor vertex and the associated weight.

  This is iterate *num_elem* time, one for each NNZ element of the row.

## 5 Results and Comparison

Here are listed some of the results obtained with the implantation. These results are compared with the given implementation done with the adjacency lists.

- **Example Directed**: from node 2
  - My implementation:
    - Size: 0.0156MB
    - Population Time: 0ms
    - BFS: 0ms
    - DFS: 0ms
  - Adjacency Lists:
    - Size: 0.0156MB
    - Population Time: 0ms
    - BFS: 0ms
    - DFS: 0ms
- **Example Undirected** (undirected graph): from node 2
  - My implementation:
    - Size: 0.0156MB
    - Population Time: 0ms
    - BFS: 0ms
    - DFS: 0ms
  - Adjacency Lists:
    - Size: 0.0156MB
    - Population Time: 0ms
    - BFS: 0ms
    - DFS: 0ms
- **Wiki Talk**: from node 2
  - My implementation:
    - Size: 153.828MB
    - Population Time: 233ms
    - BFS: 133ms
    - DFS: 140ms
  - Adjacency Lists:
    - Size: 416.117MB
    - Population Time: 731ms
    - BFS: 590ms
    - DFS: 665ms

- **Cit Patent**: from node 3774754
  - My implementation:
    - Size: 361.133MB
    - Population Time: 610ms
    - BFS: 68ms
    - DFS: 52ms
  - Adjacency Lists:
    - Size: 1181.08MB
    - Population Time: 6129ms
    - BFS: 166ms
    - DFS: 226ms
- **Dota League** (undirected graph): from node 55712
  - My implementation:
    - Size: 1595.12MB
    - Population Time: 3776ms
    - BFS: 437ms
    - DFS: 2120ms
  - Adjacency Lists:
    - Size: 4049.93MB
    - Population Time: 22339ms
    - BFS: 35161ms
    - DFS: 14951ms

When multiple iterations are done the sum of all the graphs remains the same, so it can be said that the implementation is correct and the performance usually improves in advance iteration.