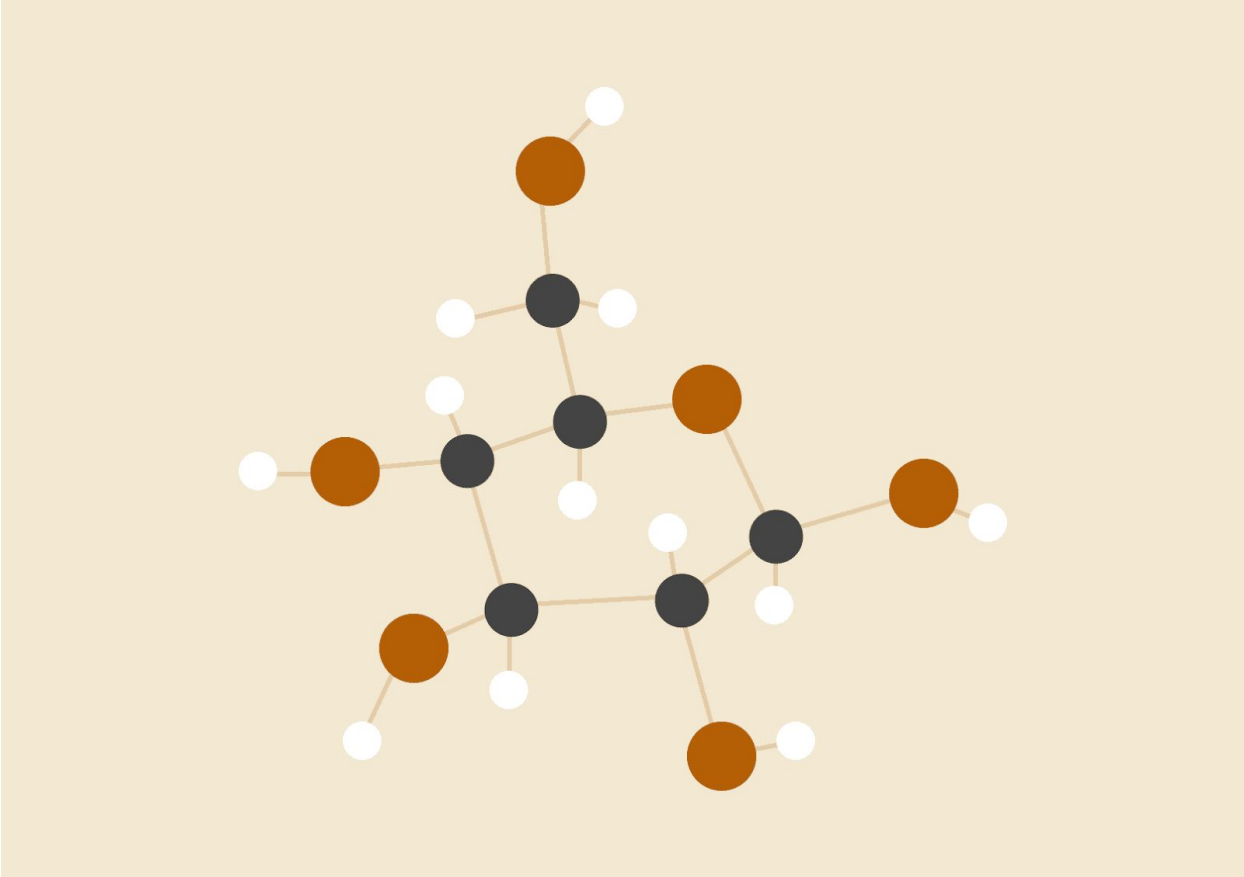
REPORT

"Simple Graph" Project in C++



Maghous Mohammed Amine

+212636393658

amine_mag@outlook.com

INTRODUCTION

This report is an overview of my work to get this project done, it contains;

- A description of the methods and their corresponding complexity.
- The data structures I used and the reason to use them.
- The algorithms I used and the reason to use them.

Used data structures:

- **std::vector<std::list<uint64_t>> adj_list;**
  - **Role:** store the graph edges, where each list contains all the vertices that can be accessed directly by the vertex that points to the position where this list is stored using a map
  - Reason:
    - I prefered to use vector because accessing is done in O(1) and because adding new vertices is usually not very frequent.
    - I used a list because adding edges is usually more frequent, on the other hand I could have used forward_list to decrease overhead since I only need a forward iterator but I frequently needed to know the size of the list, and an std::list gives me the size in O(1) not like std::forward_list where either you add an O(number_of_vertices) method to calculate the size or add a data structure to store the size and modify it every time it's changed.
- **std::map<uint64_t, T> property_values_map;**
  - **Role:** store the property values of each node/vertex where the key is the vertex ID and the value is the property value.
  - Reason: two frequent operations I use, retrieving and inserting are done in O(log(number_of_vertices)), I used instead of a vector because the number of vertices in the graph is unknown, and the ID's are not contiguos. By using this data structure I'm allocating just enough memory to hold the values.
- **std::set<uint64_t> graph_vertices_set;**
  - **Role:** It stores the vertices ID, of the vertices I already inserted in the graph
  - Reason: two frequent operations I use, retrieving and inserting are done in O(log(number_of_vertices)).
- **std::map<uint64_t, uint64_t> vertex_list_position_map;**

- **Role:** stores the position of the list in the adj_list containing directly connected vertices, where the key is the vertex Id and the values is the position.
- **Reason:** Since the graph vertices ID's are not contiguous, and these ID's can be numbers strings, chars ..etc, we can't reference directly

Functions/Methods and their description:

In red are required methods (only complexity is specified) and in blue additional methods (complexity and description are specified).

In Graph class:

- void add_edge(uint64_t source, uint64_t destination);
  - Time complexity: O(log(|V|))
    - Reason: I had to insert and retrieve values from a map and a set.
- void remove_edge(uint64_t source, uint64_t destination);
  - Time complexity: same as add_edge.
- void set_node_property(uint64_t n, T val);
  - Time complexity: O(log(|V|));
    - **Reason:** inserting in the map **property_values_map;**
- T get_node_property(uint64_t n);
  - Time complexity: O(log(|V|));
    - **Reason:** Retrieving from the map **property_values_map;**
- std::string export_node_property_to_string();
  - Time complexity: O(|V|);
    - **Reason:** iterating over the property_values_map.
  - Space complexity: O(number_of_vertices);
    - **Reason:** storing all pairs of vertices,value in a stringstream.
- std::vector<uint64_t>::size_type get_number_of_vertices();
  - Time complexity: O(1);

- uint64_t get_number_of_exiting_edges_from_source(uint64_t source_pos);
    - Time complexity: O(1)
    - **Role:** Get the number of directly connected edges to the vertex witch have it's list in position source_pos.
- bool check_weakly_connected();
    - **Role:** checks if a graph is weakly connected by comparing the set of vertices in the actual graph <<graph_vertices_set>> with the vertices visited by bfs algorithm of the corresponding undirected graph.
    - Time complexity: O(max{(|V|^2) * log(|V|), V + E})
    - Space complexity: O(|V|^2);
        - Reason: creating the undirected graph.
- void build_undirected_graph(std::vector<std::list<uint64_t>>& adj_list_undirected);
    - **Role:** creates the undirected graph from directed graph.
    - Time Complexity: O((|V|^2) * log(|V|));
    - Space Complexity: O(|V|^2);
- std::set<uint64_t> bfs_undirected_graph();
    - **Role:** iterates over the undirected graph using breadth first search using.
    - Time complexity: O(max{(|V|^2) * log(|V|), V + E})
    - Space complexity: O(|V|^2);

In graph_loader class:

- void load_graph(std::shared_ptr<graph<T>> g, std::string file_path);
    - Time complexity: O(|E|log(|E|));

In graph_algorithms class:

- bool is_weakly_connected(std::shared_ptr<graph<T>> g)
    - Time complexity: O((|V|^2) * log(|V|))
    - Space complexity: O(|V|^2);

        - The algorithm:
            - Create an undirected graph from the given directed graph
            - Take one vertex from the available ones and do a bfs starting from that vertex, and store each new visited vertex in a set
            - If at the end of the bfs algorithm the set of vertices visited is not equal to the set of all vertices in the graph then the graph is not weakly connected.
    - bool is_fully_connected(std::shared_ptr<graph<T>> g)
        - Time complexity: O(|V|).
        - The algorithm:
            -
        - Problems that might rise by using this algorithm:
            - In this algorithm I suppose that all edges where entered, correctly so if an edge was entered more than once and the graph is not fully connected, the algorithm might still give an error, one solution is to check input errors before running any algorithm.

Graph class space complexity estimation:

I've personally never heard about class space complexity, but in my understanding it will be the space occupied by the attributes (data structures) of the class in this case I've used an adjacency list with a map, so in this case only enough memory to store each edge is allocated, it's then linear on the number of edges entered.

Problems with the implementation:

- In the Sequential version
    - Using an adjacency List means that deleting vertex operation will have a complexity of O(|V|), also adding new vertices might force the vector to reallocate memory which also means copying all the data to another location.
- In the parallel version:
    - In addition to the problems stated for the sequential version, in the parallel version and exactly inside bfs_undirected_graph() method the mutex is locked at the beginning and it’s not unlocked until the thread finishes its execution so I'm not taking advantage of multithreading in that case.