Graph Representation: Adjacency List and Matrix

Introduction

In the previous post (https://algorithmtutor.com/Data-Structures/Graph/Graphs-and-Graph-Terminologies/), we introduced the concept of graphs. In this post, we discuss how to store them inside the computer. There are two popular data structures we use to represent graph: (i) Adjacency List and (ii) Adjacency Matrix. Depending upon the application, we use either adjacency list or adjacency matrix but most of the time people prefer using adjacency list over adjacency matrix.

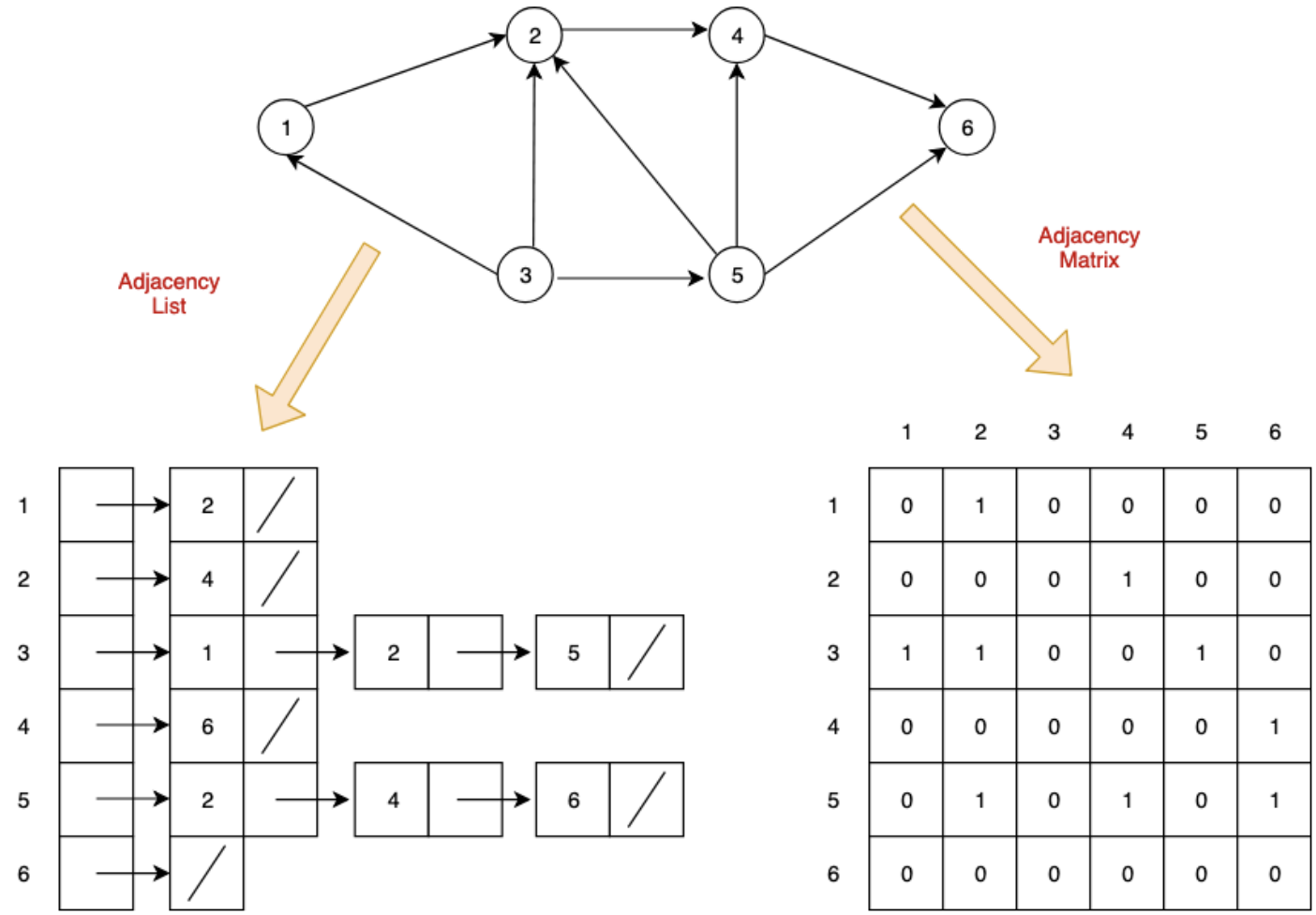
Adjacency Lists

> Adjacency lists are the right data structure for most applications of graphs.

Adjacency lists, in simple words, are the array of linked lists. We create an array of vertices and each entry in the array has a corresponding linked list containing the neighbors. In other words, if a vertex 1 has neighbors 2, 3, 4, the array position corresponding the vertex 1 has a linked list of 2, 3, and 4. We can use other data structures besides a linked list to store neighbors. I personally prefer to use a hash table and I am using the hash table in my implementation. You can also use balanced binary search trees as well. To store the adjacency list, we need $O(V + E)$ space as we need to store every vertex and their neighbors (edges).

To find if a vertex has a neighbor, we need to go through the linked list of the vertex. This requires $O(1 + deg(V))$ time. If we use balanced binary search trees, it becomes $O(1 + \log(deg(V))$ and using appropriately constructed hash tables, the running time lowers to $O(1)$.

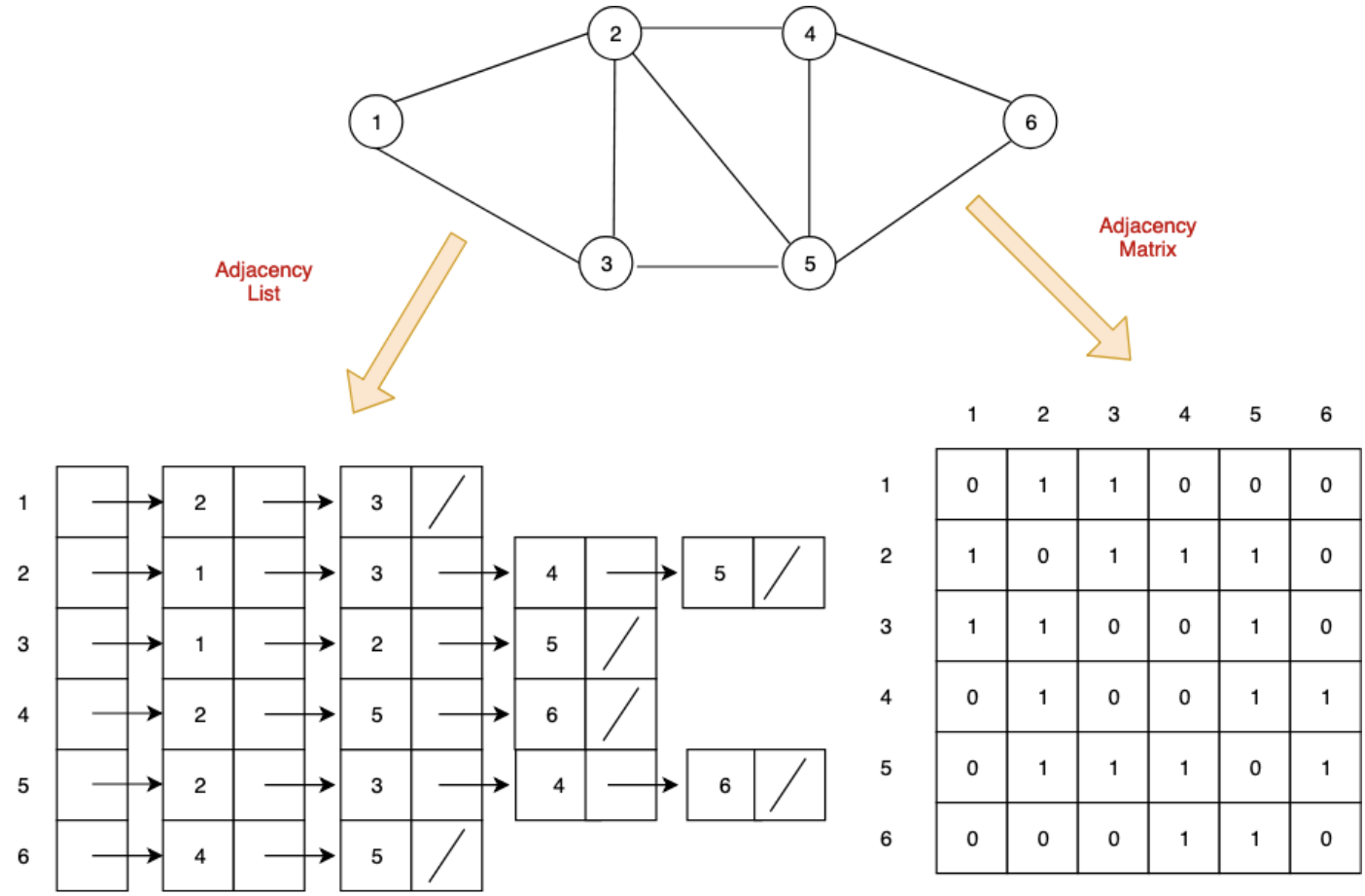
Figure 1 shows the linked list representation of a directed graph.



(

Figure 1: Adjacency List and Adjacency Matrix Representation of a Directed Graph

In an undirected graph, to store an edge between vertices A and B, we need to store B in A's linked list and vice versa. Figure 2 depicts this.



(

Figure 2: Adjacency List and Adjacency Matrix Representation of an Undirected Graph

Adjacency Matrices

An adjacency matrix is a $V \times V$ array. It is obvious that it requires $O(V^2)$ space regardless of a number of edges. The entry in the matrix will be either 0 or 1. If there is an edge between vertices A and B, we set the value of the corresponding cell to 1 otherwise we simply put 0. Adjacency matrices are a good choice when the graph is dense since we need $O(V^2)$ space anyway. We can easily find whether two vertices are neighbors by simply looking at the matrix. This can be done in $O(1)$ time. Figure 1 and 2 show the adjacency matrix representation of a directed and undirected graph.

Representing Weighted Graphs

We can modify the previous adjacency lists and adjacency matrices to store the weights. In the adjacency list, instead of storing the only vertex, we can store a pair of numbers one vertex and other the weight. Similarly, in the

adjacency matrix, instead of just storing 1 we can store the actual weight. Figure 3 illustrates this.

Loading

(

Figure 3: Adjacency List and Adjacency Matrix Representation of a Weighted Graph

Comparison

The table below summarizes the operations and their running time in adjacency list and adjacency matrix.

Operation	Adjacency List (Linked List)	Adjacency List (Hash Table)	Adjacency Matrix
Test if uv is an edge (directed)	$O(V)$	$O(1)$	$O(1)$
Test if uv is an edge (undirected)	$O(V)$	$O(1)$	$O(1)$
List v's neighbor	$O(V)$	$O(V)$	$O(V)$
List all edges	$O(V+E)$	$O(V+E)$	$O(V^2)$
Insert edge uv	$O(1)$	$O(1)$	$O(1)$

Implementation

Since I will be doing all the graph related problem using adjacency list, I present here the implementation of adjacency list only. You can find the codes in C++, Java, and Python below.

C++ Java Python

```
      // Adjacency list representation in C++
// Author: Algorithm Tutor

// std::map has running time of O(log n) for dynamic set operations.
// use std::unordered_map if you want the constant time complexity

#include
#include
#include

class Graph {
private:
    std::map<int, std::map<int, int> > graph;
public:
    void addEdge(int u, int v, int weight = 1, int isDirected = true) {
        std::map<int, std::map<int, int> >::iterator it;
        it = graph.find(u);
        if (it == graph.end()) {
            std::map<int, int> edge_map;
            edge_map[v] = weight;
            graph[u] = edge_map;

        } else {
            graph[u][v] = weight;
        }

        if (!isDirected) {
            it = graph.find(v);
            if (it == graph.end()) {
                std::map<int, int> edge_map;
                edge_map[u] = weight;
                graph[v] = edge_map;

            } else {
                graph[v][u] = weight;
            }
        }
    }

    void printGraph() {
        std::map<int, std::map<int, int> >::iterator it;
        std::map<int, int>::iterator it2;
        for (it = graph.begin(); it != graph.end(); ++it) {
            std::cout<first<<": ";
            for (it2 = it->second.begin(); it2 != it->second.end(); ++it2) {
                std::cout<<"("<first<<","<second<<")";
                std::cout<<"   ";
            }
            std::cout<<std::endl;
```

```
        }
    }
};

int main() {
    Graph g;
    g.addEdge(1, 2, 7, false);
    g.addEdge(1, 3, 2, false);
    g.addEdge(2, 3, 1, false);
    g.addEdge(2, 4, 5, false);
    g.addEdge(2, 5, 3, false);
    g.addEdge(3, 5, 11, false);
    g.addEdge(4, 5, 10, false);
    g.addEdge(4, 6, 7, false);
    g.addEdge(5, 6, 4, false);
    g.printGraph();
    return 0;
}
```

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (n.d.). Introduction to algorithms (3rd ed.). The MIT Press.
2. Jeff Erickson. Algorithms (Prepublication draft). http://algorithms.wtf (http://algorithms.wtf)
3. Steven S. Skiena. 2008. The Algorithm Design Manual (2nd ed.). Springer Publishing Company, Incorporated.

adjacency list (/tags/adjacency-list/) adjacency matrix (/tags/adjacency-matrix/)
graphs (/tags/graphs/) graphs representation (/tags/graphs-representation/)

‹ Graphs and Graph Terminologies (/Data-Structures/Graph/Graphs-and-Graph-Terminologies/)