Universidade Federal de Ouro Preto
Lecture Notes
Graphs

Prof. Rodrigo Silva

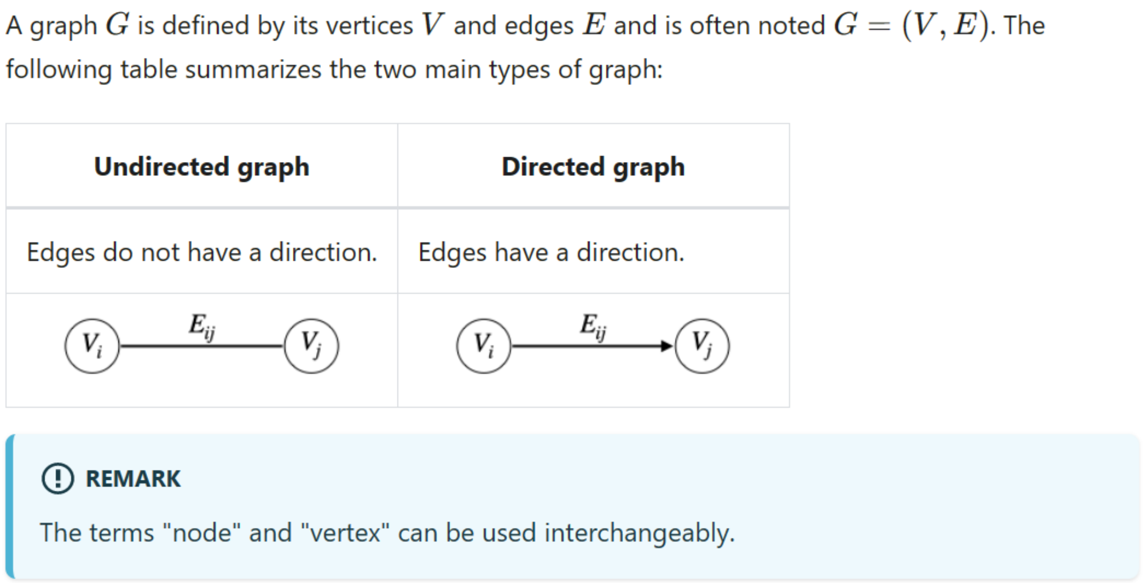
April 3, 2023

Source

- Super Study Guide - Graphs (`https://superstudy.guide/algorithms-data-structures/graphs-trees/graphs`)

1 Graphs

1.1 Definition



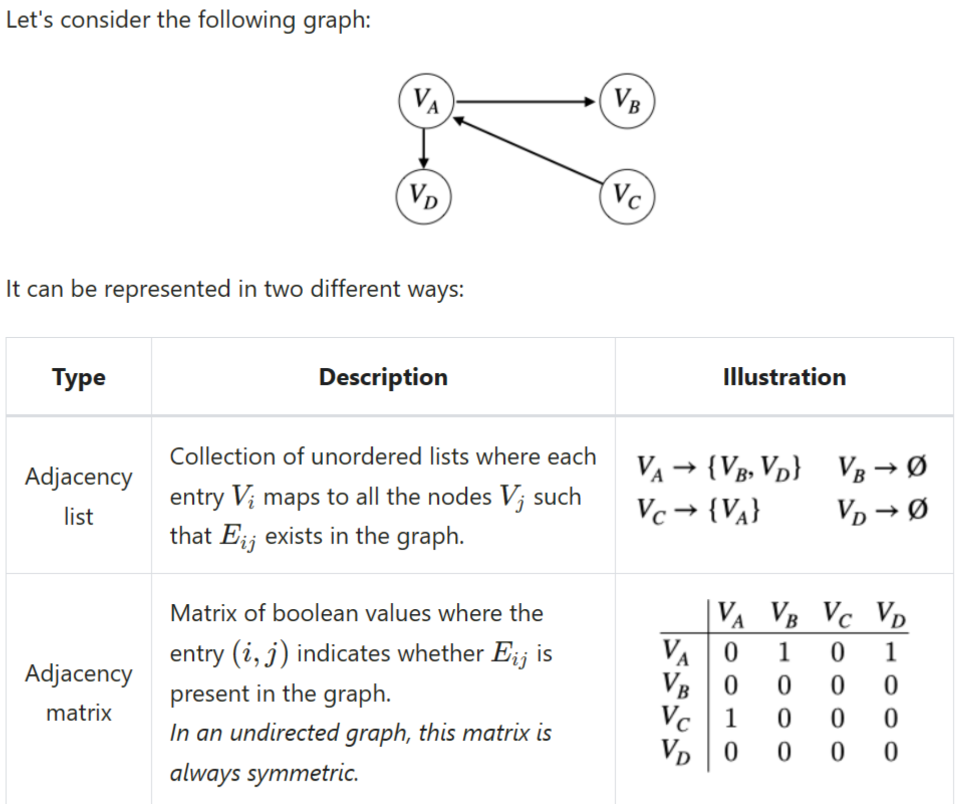
A graph G is defined by its vertices V and edges E and is often noted $G = (V, E)$. The following table summarizes the two main types of graph:

Undirected graph	**Directed graph**
Edges do not have a direction.	Edges have a direction.
$V_i \overset{E_{ij}}{\text{———}} V_j$	$V_i \xrightarrow{E_{ij}} V_j$

ⓘ **REMARK**

The terms "node" and "vertex" can be used interchangeably.

1.2 Representation



Let's consider the following graph:

It can be represented in two different ways:

Type	Description	Illustration
Adjacency list	Collection of unordered lists where each entry V_i maps to all the nodes V_j such that E_{ij} exists in the graph.	$V_A \to \{V_B, V_D\}$ $V_B \to \emptyset$ $V_C \to \{V_A\}$ $V_D \to \emptyset$
Adjacency matrix	Matrix of boolean values where the entry (i, j) indicates whether E_{ij} is present in the graph. *In an undirected graph, this matrix is always symmetric.*	$\begin{array}{c\|cccc} & V_A & V_B & V_C & V_D \\ \hline V_A & 0 & 1 & 0 & 1 \\ V_B & 0 & 0 & 0 & 0 \\ V_C & 1 & 0 & 0 & 0 \\ V_D & 0 & 0 & 0 & 0 \end{array}$

2 Graphs in C++

2.1 Adjacency list

```cpp
#include <iostream>
#include <unordered_map>
#include <vector>
#include <stack>
#include <algorithm>

using namespace std;

class Graph {
private:
    unordered_map<string, vector<pair<string, int>>> adj_list;

public:
    void add_node(string node) {
        if (adj_list.find(node) == adj_list.end()) {
            adj_list[node] = {};
        }
    }

    void add_edge(string node1, string node2, int weight = 1) {
        add_node(node1);
        add_node(node2);
        adj_list[node1].push_back({node2, weight});
        adj_list[node2].push_back({node1, weight});
    }

    void remove_edge(string node1, string node2) {
        auto it = find_if(adj_list[node1].begin(), adj_list[node1].end(),
                          [node2](const pair<string, int>& p) {
                              return p.first == node2;
                          });
        if (it != adj_list[node1].end()) {
            adj_list[node1].erase(it);
        }
        it = find_if(adj_list[node2].begin(), adj_list[node2].end(),
                     [node1](const pair<string, int>& p) {
                         return p.first == node1;
                     });
        if (it != adj_list[node2].end()) {
            adj_list[node2].erase(it);
        }
    }

    void remove_node(string node) {
        adj_list.erase(node);
        for (auto& [other_node, neighbours] : adj_list) {
            auto it = find_if(neighbours.begin(), neighbours.end(),
                              [node](const pair<string, int>& p) {
                                  return p.first == node;
                              });
            if (it != neighbours.end()) {
                neighbours.erase(it);
            }
        }
    }

    bool is_path(string node1, string node2) {
        unordered_set<string> visited;
        stack<string> st;
        st.push(node1);
        while (!st.empty()) {
            string node = st.top();
            st.pop();
            if (node == node2) {
                return true;
            }
            if (visited.find(node) == visited.end()) {
                visited.insert(node);
                for (auto& [neighbour, weight] : adj_list[node]) {
                    st.push(neighbour);
                }
            }
        }
        return false;
    }
};

int main() {
    Graph g;
    g.add_edge("A", "B"); // add an edge between nodes ’A’ and ’B’
    g.add_edge("B", "C"); // add an edge between nodes ’B’ and ’C’
    for (auto& [node, neighbours] : g.adj_list) {
        cout << node << ": ";
        for (auto& [neighbour, weight] : neighbours) {
            cout << "(" << neighbour << ", " << weight << ") ";
        }
        cout << endl;
    }
    // Output: A: (B, 1)
    //         B: (A, 1) (C, 1)
    //         C: (B, 1)

    g.add_edge("A", "D"); // add an edge between nodes 'A' and 'D'
    for (auto& [node, neighbours] : g.adj_list) {
        cout << node << ": ";
        for (auto& [neighbour, weight] : neighbours) {
            cout << "(" << neighbour << ", " << weight << ") ";
        }
        cout << endl;
    }
}
```

Listing 1: Example 1 - Graph as Adjacency List C++ (Iteractive)

2.2 Adjacency matrix

```cpp
#include <iostream>
#include <vector>
using namespace std;

class Graph {
public:
    Graph(int num_nodes) : num_nodes(num_nodes), adj_matrix(num_nodes, vector<int>(num_nodes, 0)) {}

    void add_node() {
        num_nodes++;
        for (auto& row : adj_matrix) {
            row.push_back(0);
        }
        adj_matrix.push_back(vector<int>(num_nodes, 0));
    }

    void add_edge(int node1, int node2, int weight = 1) {
        adj_matrix[node1][node2] = weight;
        adj_matrix[node2][node1] = weight;
    }

    void remove_edge(int node1, int node2) {
        adj_matrix[node1][node2] = 0;
        adj_matrix[node2][node1] = 0;
    }

    void remove_node(int node) {
        num_nodes--;
        adj_matrix.erase(adj_matrix.begin() + node);
        for (auto& row : adj_matrix) {
            row.erase(row.begin() + node);
        }
    }

    bool is_path(int node1, int node2) {
        vector<bool> visited(num_nodes, false);
        vector<int> queue;
        queue.push_back(node1);
        visited[node1] = true;

        while (!queue.empty()) {
            int current_node = queue.front();
            queue.erase(queue.begin());
            if (current_node == node2) {
                return true;
            }
            for (int neighbor = 0; neighbor < num_nodes; neighbor++) {
                int weight = adj_matrix[current_node][neighbor];
                if (weight > 0 && !visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.push_back(neighbor);
                }
            }
        }

        return false;
    }

private:
    int num_nodes;
    vector<vector<int>> adj_matrix;
};

int main() {
    Graph g(3);  // create a graph with 3 nodes
    g.add_edge(0, 1);  // add an edge between nodes 0 and 1
    g.add_edge(1, 2);  // add an edge between nodes 1 and 2
    for (auto row : g.adj_matrix) {
        for (auto val : row) {
            cout << val << " ";
        }
        cout << endl;
    }  // print the adjacency matrix
    // Output: 0 1 0
    //         1 0 1
    //         0 1 0

    g.add_node();  // add a node to the graph
    g.add_edge(0, 3);  // add an edge between nodes 0 and 3
    for (auto row : g.adj_matrix) {
        for (auto val : row) {
            cout << val << " ";
        }
        cout << endl;
    }  // print the adjacency matrix
    // Output: 0 1 0 1
    //         1 0 1 0
    //         0 1 0 0
    //         1 0 0 0

    g.remove_edge(0, 3);  // remove the edge between nodes 0 and 3
    g.remove_node(2);  // remove node 2 from the graph
    for (auto row : g.adj_matrix) {
        for (auto val : row) {
            cout << val << " ";
        }
        cout << endl;
    }  // print the adjacency matrix
}
```

Listing 2: Example 1 - Graph as Adjacency List C++

3 Graphs in Python

3.1 Adjacency list

```
class Graph:
    def __init__(self):
        self.adj_list = {}
```

```python

    def add_node(self, node):
        if node not in self.adj_list:
            self.adj_list[node] = []

    def add_edge(self, node1, node2, weight=1):
        self.add_node(node1)
        self.add_node(node2)
        self.adj_list[node1].append((node2, weight))
        self.adj_list[node2].append((node1, weight))

    def remove_edge(self, node1, node2):
        for i, (node, weight) in enumerate(self.adj_list[node1]):
            if node == node2:
                del self.adj_list[node1][i]
                break
        for i, (node, weight) in enumerate(self.adj_list[node2]):
            if node == node1:
                del self.adj_list[node2][i]
                break

    def remove_node(self, node):
        del self.adj_list[node]
        for other_node in self.adj_list:
            for i, (n, w) in enumerate(self.adj_list[other_node]):
                if n == node:
                    del self.adj_list[other_node][i]
                    break

    def is_path(self, node1, node2):
        visited = set()
        stack = [node1]
        while stack:
            node = stack.pop()
            if node == node2:
                return True
            if node not in visited:
                visited.add(node)
                stack.extend(neighbour for neighbour, weight in self.adj_list[node])
        return False

# example usage
g = Graph()
g.add_edge('A', 'B')  # add an edge between nodes 'A' and 'B'
g.add_edge('B', 'C')  # add an edge between nodes 'B' and 'C'
print(g.adj_list)  # print the adjacency list
# Output: {'A': [('B', 1)], 'B': [('A', 1), ('C', 1)], 'C': [('B', 1)]}

g.add_edge('A', 'D')  # add an edge between nodes 'A' and 'D'
print(g.adj_list)  # print the adjacency list
# Output: {'A': [('B', 1), ('D', 1)], 'B': [('A', 1), ('C', 1)], 'C': [('B', 1)], 'D': [('A', 1)]}

g.remove_edge('A', 'D')  # remove the edge between nodes 'A' and 'D'
g.remove_node('B')  # remove node 'B' from the graph
print(g.adj_list)  # print the adjacency list
# Output: {'A': [], 'C': []}

print(g.is_path('A', 'C'))  # check if a path exists between nodes 'A' and 'C'
# Output: False
print(g.is_path('A', 'D'))  # check if a path exists between nodes 'A' and 'D'
# Output: False
print(g.is_path('D', 'A'))  # check if a path exists between nodes 'D' and 'A'
```

```python
# Output: True
```

Listing 3: Example 1 - Graph as Adjacency List Python

3.2 Adjacency matrix

```python
class Graph:
    def __init__(self, num_nodes):
        self.num_nodes = num_nodes
        self.adj_matrix = [[0 for _ in range(num_nodes)] for _ in range(num_nodes)]

    def add_node(self):
        self.num_nodes += 1
        for row in self.adj_matrix:
            row.append(0)
        self.adj_matrix.append([0 for _ in range(self.num_nodes)])

    def add_edge(self, node1, node2, weight=1):
        self.adj_matrix[node1][node2] = weight
        self.adj_matrix[node2][node1] = weight

    def remove_edge(self, node1, node2):
        self.adj_matrix[node1][node2] = 0
        self.adj_matrix[node2][node1] = 0

    def remove_node(self, node):
        self.num_nodes -= 1
        self.adj_matrix.pop(node)
        for row in self.adj_matrix:
            row.pop(node)

    def is_path(self, node1, node2):
        visited = [False] * self.num_nodes
        queue = [node1]
        visited[node1] = True

        while queue:
            current_node = queue.pop(0)
            if current_node == node2:
                return True
            for neighbor, weight in enumerate(self.adj_matrix[current_node]):
                if weight > 0 and not visited[neighbor]:
                    visited[neighbor] = True
                    queue.append(neighbor)

        return False


# example usage
g = Graph(3)  # create a graph with 3 nodes
g.add_edge(0, 1)  # add an edge between nodes 0 and 1
g.add_edge(1, 2)  # add an edge between nodes 1 and 2
print(g.adj_matrix)  # print the adjacency matrix
# Output: [[0, 1, 0], [1, 0, 1], [0, 1, 0]]

g.add_node()  # add a node to the graph
g.add_edge(0, 3)  # add an edge between nodes 0 and 3
print(g.adj_matrix)  # print the adjacency matrix
# Output: [[0, 1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0]]

g.remove_edge(0, 3)  # remove the edge between nodes 0 and 3
g.remove_node(2)  # remove node 2 from the graph
print(g.adj_matrix)  # print the adjacency matrix
# Output: [[0, 1, 1], [1, 0, 0], [1, 0, 0]]

```

```python
print(g.is_path(0, 2))  # check if a path exists between nodes 0 and 2
# Output: True
print(g.is_path(0, 1))  # check if a path exists between nodes 0 and 1
# Output: True
print(g.is_path(1, 2))  # check if a path exists between nodes 1 and 2
# Output: False
```

Listing 4: Example 1 - Graph as Matrix Python