Graph Abstract Data Type

A graph is an abstract data structure that consists of a set of vertices (or nodes) and edges. All vertices in a graph are connected to each other through a set of edges. Both the vertices and the edges of a graph can contain data. We can traverse a graph, we can find spanning trees in a graph, we can use a graph to represent a data flow, …

C++26

At the time of writing, there is no support for graphs in the C++ standard library. There is, however, a proposal that plans to add this (P1709) in C++26. Because there is no official implementation yet (There is a preliminary, non-official implementation here: https://github.com/pratzl/graph), we will write our own in this lab.

Graph Types

Before we can begin to implement a graph, we need to know what types of graphs there are, below, we give a short summary.

Directed vs Undirected graphs:

If an edge has a direction (From … To …) we say the graph is directed, if this is not the case, the graph is undirected.

Cycles

If it is not allowed for cycles to occur in a graph, we call the graph “acyclic”.

Multi-graphs

If it’s allowed for multiple edges between the same two nodes to exist, we call a graph a multi-graph. If this is not allowed, we call a graph a simple graph.

Dense vs Sparse Graphs

A graph can also be dense or sparse. Density or Sparsity refers to the ratio between the number of edges and the number of nodes in a graph. A graph with little nodes, or many edges is called a dense graph, while a graph with many nodes, or little edges is called a sparse graph.

Graph Representations

There are multiple ways of representing the edges of a graph, the three most important ones are:

- Adjacency Matrix
- Adjacency List
- Incidence Matrix

To describe edges, we will number all nodes in a graph.

Adjacency Matrix

In an adjacency matrix, the rows of each matrix represent the “from” dimension, while the columns represent the “to” dimension. If an edge is present in a graph, then the corresponding element in the matrix will be set to 1. If this is not the case, the element in the matrix is 0. It is also possible to swap the “from” and “to” dimensions in an adjacency matrix. In some applications, this is the preferred layout. If a graph is undirected, every edge will be represented as 1 edge in both directions. For example, the undirected edge between nodes 3 and 4 will be represented as 2 edges: 1 from 3 to 4, and 1 from 4 to 3.

Adjacency List

In an adjacency list, we keep a list for every vertex. This list then contains all the neighbors of a specific vertex. Each element of this list thus describes a single edge.

Incidence Matrix

In an incidence matrix, every row represents a vertex, and every column represents an edge. In this matrix, a 1 is placed on the rows that are the endpoints of an edge. If a graph is directed, -1 and 1 are used to indicate the direction of an edge. For example, if we have an undirected graph with 3 nodes: 1, 2 and 3 and 2 edges: 1-2 and 2-3, then the following elements will be 1, while all others will be 0:

- (0,0) - Edge 1-2
- (1,0) – Edge 1-2
- (1,1) – Edge 2-3
- (2,1) – Edge 2-3

Code

In this lab, we will use the following interface to operate on graphs:

```cpp
// Class that represents a graph with weighted edges.
// The graph is represented as an adjacency matrix.
// The weights of the edges are stored in the matrix as integers.
// A weight of -1 means that the edge does not exist.
class Graph {
public:
    // Constructor
    explicit Graph(std::size_t node_count);

    // Returns the number of nodes in the graph
    std::size_t num_nodes();

    // Returns every edge in the graph
    std::vector<std::pair<int, int>> edges();

    // Returns all edges that are connected to a given node
    std::vector<std::pair<int, int>> connected_edges(int node);

    // Returns true if the given edge exists in the graph, false otherwise
    bool exists(std::pair<int, int> edge);

    // Returns a reference to the weight of a given edge
    // Can be used to get or set the weight of an edge
    int& operator[](std::pair<int, int> edge);

private:
    // The adjacency matrix of the graph
    std::vector<std::vector<int>> adjacency_matrix;
};
```

This graph has weighted edges, this means that every edge also has an associated weight, which indicates how “long” or “costly” a certain edge is. Initially, we’ll use a weight of -1 to indicate the absence of an edge, and a weight of 1 to indicate the presence of an edge, but later on, we’ll use different weights.

1. Write an empty class that fulfills this interface. You can have each method return a dummy value for now.
2. For every method in the given interface, write at least 1 unit test that demonstrates the correct operation of that method.
3. Now correctly implement the graph class using an adjacency matrix. You can use the tests you wrote earlier to verify that your class works correctly.
4. The Dijkstra algorithm is an algorithm that is used to find the shortest path between two vertices. Implement this algorithm for your graph class, using the interface you can find below.

5. The A* algorithm is a modified version of the Dijkstra algorithm. Make a copy of the code of your Dijkstra implementation, and use it as a starting point for implementing A*.

```
// Algorithm takes a reference to a weighted graph, a start node, and an end node,
// and returns a sequence (vector) of nodes that represent the shortest (least-cost) path from the start to the end,
// according to the weights of each edge.
// start and end should both be included in the returned path.
std::vector<int> dijkstra(Graph& g, int start, int end);

// Algorithm takes a reference to a weighted graph, a start node, an end node, and a reference to a heuristic function h,
// and returns a sequence (vector) of nodes that represent the shortest (least-cost) path from the start to the end,
// according to the weights of each edge and the heuristic function.
// The heuristic function should take a node and a reference to the graph, and return an integer.
// start and end should both be included in the returned path.
std::vector<int> a_star(Graph& g, int start, int end, const std::function<int(int, Graph&)>& h);
```

Report

1. When would you use which graph representation? Why?
2. In the lab on binary trees, we implemented different tree-traversal algorithms (Depth-first and Breadth-first), we can also implement these algorithms for a graph, but we need to make some changes to the algorithm. Which changes?
3. What are the differences between A* and Dijkstra? Is one algorithm better than the other? Why?