

Weighted graphs and their implementations in C++

Q: How does a weighted graph differ from an ordinary graph?

A: In a weighted graph each edge (x,y) has associated with it a **weight** $w((x,y))$, which is a number.

Common interpretations of weights

- graph represents a road map and nodes are locations or intersections on the map: $w((x,y))$ can be the distance from x to y or the time required to get from x to y

- graph represents a project and the nodes are different tasks in the project: $w((x,y))$ is the time it takes to complete the task x

- graph represents a pipe network under construction and the nodes represent junctions in the pipe network: $w((x,y))$ can be the cost of constructing the pipe from x to y

Common use

- find the shortest/fastest/cheapest path from some node to some other node

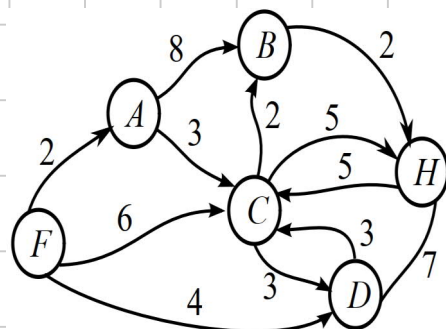
- BFS assumes $w((x,y)) = 1$ for all edges

Example

Goal: shortest path from F to H .

One BFS solution: $\langle F, D, H \rangle$

True solution: $\langle F, A, C, B, H \rangle$



Implementation 1

For each node y adjacent to x , store pair $\langle y, w((x,y)) \rangle$

```
struct Node
{
    // All the data stored in the node
    int id;
    std::string name;
    // ...

    std::vector<std::pair<Node*,Cost>> to_neighbours;
};
```

Suitable when

- only need to move forward along edges
- edges added or deleted infrequently
- only data associated with edge is cost (weight)

Implementation 2

```
struct Node
{
    // All the data stored in the node
    int id;
    std::string name;
    // ...

    // ...map, not set!
    std::unordered_map<Node*,Cost> to_neighbours;
};
```

Suitable when

- only need to move forward along edges
- edges added or deleted frequently
- only data associated with edge is cost (weight)

Implementation 3

```
struct Edge
{
    int cost;
    string name;
    // ...
};

struct Node
{
    // All the data stored in the node
    int id;
    std::string name;
    // ...

    // ...map, not set!
    std::unordered_map<Node*,Edge> to_neighbours;
};
```

Suitable when

- only need to move forward along edges
- edges added or deleted frequently
- there is much data associated with edge

Implementation 4

```
// In undirected graphs, edge data can be shared between directions
struct Edge
{
    int cost;
    string name;
    HugeData data;
    // ... too much data or changing data
};

struct Node
{
    // All the data stored in the node
    int id;
    std::string name;
    // ...

    // ...map, not set!
    std::unordered_map<Node*, std::shared_ptr<Edge>> to_neighbours;
};
```

Suitable when

- only need to move forward along edges
- edges added or deleted frequently
- there is much data associated with edge
- the graph is undirected and we do not wish to keep two copies of the same edge

Tämä teos on lisensoitu Creative Commons Nimeä-EiKaupallinen-EiMuutoksia 4.0 Kansainvälinen -lisenssillä. Tarkastele lisenssiä osoitteessa <http://creativecommons.org/licenses/by-nc-nd/4.0/>.

tekijä: Frank Cameron

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>.

made by Frank Cameron

