Topological Sorting

Topological Sorting for a Directed Acyclic Graph (*DAG*) is a linear ordering of vertices, such that for every edge from vertex *u* to *v*, *u* comes before *v*. It is a graph traversal in which every vertex *a* is visited only after all its dependencies are visited. For example, topological sorting may be used to represent an order of tasks to be performed, if every vertex represents a task and the edges represent a constraint that one task must be performed before another.

Using DFS:

DFS is somewhat modified to get a topological ordering of the vertices. This algorithm is started from a vertex and then the method is recursively called for all its adjacent vertices. A temporary stack is used. For a vertex, after all its adjacent vertices, only then, it is pushed in the stack. Later, the stack is emptied by popping the vertices one by one, which is, ultimately, the topological sorting of the DAG. Consider an edge uv directed from u to v. If the vertex v occurs before u in the topological ordering, the graph is not an DAG.

Below is a implementation of the same in `topoSortUsingDFS.cpp`. It uses the `class Graph` as a graph from `graph.hpp` mentioned before.

```
// topoSortUsingDFS.cpp

#include "graph.hpp"

/**
 * @brief Helps topologicalSort by recursively applying DFS. Reversed topological
ordering is stored in
 * order
 *
 * @param G The Graph Object.
 * @param u The vertex on which DFS is to be performed.
 * @param order The vector container in which reverse topological ordering is stored.
 * @param visited The vector boolean container to keep track of visited vertices.
 */
void
topoSortUtil(const Graph &G, Vertex u, std::vector<Vertex> &order, std::vector<bool>
&visited) {
    visited[u] = true;
    const std::vector<Vertex> & adj = G.getAdj(u);
    for (Vertex v : adj) {
        if (not visited[v])
            topoSortUtil(G, v, order, visited);
    }
    order.push_back(u);
}


/**
 * @brief Prints a topological ordering of the vertices if the Graph
 * is an DAG. Prints an error message if the Graph is not an DAG.
```

```
 *
 * @param G The Graph object
 * @param err Default = "IMPOSSIBLE". The error message to be prited
 * if the graph is not an DAG.
 */
void
topologicalSort(const Graph &G, std::string err = "IMPOSSIBLE") {
    int V = G.getV();
    int pos[V + 1]={};
    std::vector<Vertex> topologicalOrdering;
    topologicalOrdering.reserve(V);
    std::vector<bool> visited(V + 1);

    for (Vertex i = Vertex(1); i <= V; ++i) {
        if (not visited[i])
            topoSortUtil(G, i, topologicalOrdering, visited);
    }

    std::reverse(topologicalOrdering.begin(), topologicalOrdering.end());

    int p = 0;
    for (Vertex u : topologicalOrdering) {
        pos[u] = p; ++p;
    }

    for (Vertex i = Vertex(1); i <= V; ++i) {
        const std::vector<Vertex> &adj = G.getAdj(i);
        for (Vertex v : adj) {
            if (pos[v] < pos[i]) {
                std::cout << err << "\n";
                return;
            }
        }
    }

    for (int u : topologicalOrdering) {
        std::cout << u << ' ';
    }
    std::cout << "\n";
}
```

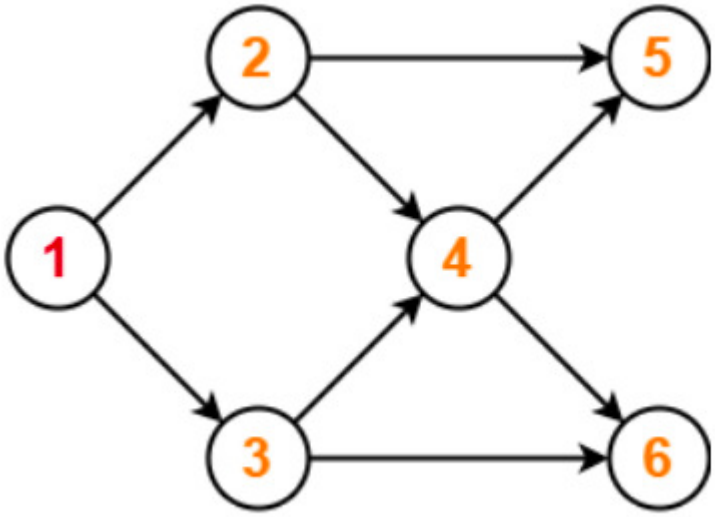
```
#include "graph.hpp"

void topologicalSort(const Graph &G, std::string err = "IMPOSSIBLE");

int main() {
    Graph G(6, true);
    G.addEdge(1, 2);
    G.addEdge(1, 3);
    G.addEdge(2, 4);
    G.addEdge(2, 5);
    G.addEdge(3, 4);
    G.addEdge(3, 6);
```

```
    G.addEdge(4, 5);
    G.addEdge(4, 6);
    topologicalSort(G);
    return 0;
}
```

The example graph used in the `main.cpp` file is:



After compiling and running,

```
$ g++ topoSortUsingDFS.cpp main.cpp -std=gnu++17
$ ./a.out
1 3 2 4 6 5
```

- Time Complexity: $O(V + E)$
- Space Complexity: $O(V)$

Since this algorithm is a modification of the DFS algorithm with an extra temporary stack, the time complexity is the same. It needs extra space for the stack.

Kahn's Algorithm:

Kahn's algorithm is based on including vertices with no incoming edges and eliminating vertices with no outgoing edges.

In this algorithm, a queue is maintained. Initially, the queue is empty. Also, a count of in-degree of all vertices is maintained. For every outgoing edge, the in-degree of the destination vertex is incremented.

Then, all vertices with no incoming edge or with in-degree equal to 0 is enqueued. Then, until the queue is not empty, following steps are performed:

- A vertex from the queue is dequeued.
- For every such vertex, the in-degrees' of its adjacent vertices are decremented.
- After decrementing, if the in-degree becomes equal to 0, the adjacent vertex is enqueued in the queue.

Below is a implementation of the same in `topoSortUsingKahn.cpp`. It uses the `class Graph` as a parameter from `graph.hpp` mentioned before.

```
// topoSortUsingKahn.cpp

#include "graph.hpp"

void
topologicalSort(const Graph &G, const std::string err = "IMPOSSIBLE") {
    int V = G.getV();
    Vertex inDegree[V + 1]={};
    std::queue<Vertex> Q;
    std::vector<Vertex> topologicalOrdering;
    topologicalOrdering.reserve(V);

    // Count incoming edges for all vertices
    for (Vertex u = Vertex(1); u <= V; ++u) {
        const std::vector<Vertex> & adj = G.getAdj(u);
        for (Vertex v : adj) {
            inDegree[v]++;
        }
    }

    for (Vertex u = Vertex(1); u <= V; ++u) {
        // Enqueue the vertex with 0 in-degree
        if (inDegree[u] == 0)
            Q.push(u);
    }

    while (not Q.empty()) {
        Vertex u = Q.front();
        Q.pop();

        topologicalOrdering.push_back(u);

        const std::vector<Vertex> & adj = G.getAdj(u);
        for (Vertex v : adj) {

            // Decrement in-degree of adjacent vertex
            --inDegree[v];

            // Enqueue the vertex with 0 in-degree
            if (inDegree[v] == 0)
                Q.push(v);
        }
    }

    if (Vertex(topologicalOrdering.size()) != V) {
        std::cout << err << "\n";
        return;
    }

    for (int u : topologicalOrdering) {
        std::cout << u << ' ';
    }
```

```
    std::cout << "\n";
}
```

The same `main.cpp` is used and the same example diagram shown before in the section *Using DFS* of *Topological Sort*.

After compiling and running `main.cpp`,

```
$ g++ topoSortUsingKahn.cpp main.cpp -std=gnu++17
$ ./a.out
1 2 3 4 5 6
```

- Time Complexity: $O(V + E)$
- Space Complexity: $O(V)$

Dijkstra

Given a directed or undirected weighted graph with V vertices and E edges with no negative cycles, the problem is to find the lengths of the shortest paths from a starting vertex s to all other vertices, and find the shortest paths themselves. This problem is called *single-source shortest paths problem*.

We maintain an array $d[]$ where for every vertex u, $d[u]$ is the length of the shortest path from s to u. We initialize $d[u] = \infty \ \forall \ u \in V - \{s\}$ where ∞ is some large number. Also, $d[s] = 0$.

We also maintain an array $p[]$ which keeps track of immediate predecessor, i.e., $p[v]$ would store the vertex which is predecessor of v. This array would be helpful in restoring the path from source vertex s to any other vertex t.

The algorithm runs for almost V iterations.
At each iteration, an unmarked vertex u with least value $d[u]$ is extracted and marked.
From vertex u, relaxations are performed on all vertices adjacent to it. That is, for each adjacent vertex v, we minimize the value of $d[v]$. If the current edge length is len, then

$$d[v] = min(d[v], len + d[u])$$

On every relaxation we perform, we update the value of $p[v]$ as:

$$p[v] = u$$

The time for extracting an unmarked vertex u with least value $d[u]$ is naively $O(V)$ if we just loop through all vertices. The step of extraction is performed V times for every vertex. Since the time for relaxation of an edge is simply $O(1)$ and we perform the step of relaxation E times for every edge. The time complexity of this algorithm would be $O(V^2 + E)$.

We can use data structures such as AVL trees, binary heap, etc. to solve this problem of extracting minimum in $O(\log V)$. For example, we can make a compromise and use a binary heap for both operations of extraction (in $O(\log V)$) and updating the value after relaxation (in $O(\log V)$).
Then, the total complexity would be,
$O(V \log V + E \log V) = O((V + E) \log V) \approx O(E \log V)$.

Here is an implementation for the algorithm using the `graph.hpp` header introduced before. The file `dijkstra.hpp` has `struct Dijkstra` inherited publicly from `class weightedGraph` in `graph.hpp`.

```
// dijkstra.hpp

#ifndef DIJKSTRA_HPP
#define DIJKSTRA_HPP

#include "graph.hpp"

struct Dijkstra : public weightedGraph {
    using ll = long long int;
    const ll INF = 0x3f3f3f3f3f3f3f3f;

    ll *d;
    Vertex *p;
```

```
    /**
     * @brief Constructs a new Dijkstra object.
     *
     * @param _V Number of vertices.
     * @param _directed Default = false. specify true if graph is directed.
     */
    Dijkstra(Vertex _V, bool _directed = false) : weightedGraph(_V, _directed) {
        d = new ll[V + 1]();
        p = new Vertex[V + 1]();
    }

    /**
     * @brief Constructs a new Dijkstra object from a weighted Graph object.
     *
     * @param G The weighted Graph Object to be copied.
     */
    Dijkstra(weightedGraph G) : weightedGraph(G) {
        d = new ll[V + 1];
        p = new Vertex[V + 1]();
    }

    /**
     * @brief Constructs a new Dijkstra object from an old one.
     * It calls the weightedGraph copy constructor.
     *
     * @param copy The old Dijkstra object to be copied.
     */
    Dijkstra(const Dijkstra &copy) : weightedGraph(copy) {
        memcpy(d, copy.d, sizeof(ll) * (V + 1));
    }


    /**
     * @brief Destroys the Dijkstra object.
     *
     */
    ~Dijkstra() {
        delete []d;
    }

    /**
     * @brief finds the lengths of shortest paths from the source vertex to all
     * vertices and stores in Dijkstra::d[].
     *
     * @param s Default = 1. The source vertex.
     */
    void solveShortestPaths(Vertex s = 1) {
        using length = std::pair<ll, Vertex>;
        std::priority_queue<length, std::vector<length>, std::greater<length>>
                                                                        pq;
        memset(d, 0x3f, sizeof(ll) * (V + 1)); // initializing d[] to INF
        d[s] = 0LL;
        p[s] = -1;
        pq.push(length(d[s], s));
        while (not pq.empty()) {
```

```
            Vertex u = pq.top().second;
            ll dist = pq.top().first;
            pq.pop();

            if (dist > d[u])
                continue;

            for (Edge e : adj[u]) {
                auto[v, len] = e;
                if (len + d[u] < d[v]) {
                    d[v] = len + d[u];
                    p[v] = u;
                    pq.push(length(d[v], v));
                }
            }
        }
    }

    /**
     * @brief prints a path from start vertex s to destination vertex t. Also
     * returns a vector container having the path.
     *
     * @param s Start vertex s.
     * @param t Destination vertex t.
     * @return std::vector<Vertex> Vector Container having the correct order
     * of vertices in its path.
     */
    std::vector<Vertex> printPath(Vertex s, Vertex t) {
        std::vector<Vertex> path;
        path.reserve(V);
        for (Vertex u = t; u != s; u = p[u])
            path.push_back(u);
        path.push_back(s);
        std::reverse(path.begin(), path.end());
        for (int u : path) {
            std::cout << u << ' ';
        }
        std::cout << "\n";
        return path;
    }

};
```

The C++ STL `std::priority_queue` uses a binary heap. Therefore, the push and pop operations would be $O(\log V)$.
Hence, the above function `solveShortestPaths(s)` runs in $O(E \log V)$.

The process of printing is also very simple. The idea is to store the parent of destination vertex u, $p[u]$ in a container and assign u as $p[u]$. The function `printPath(s, t)` runs in $O(V)$.