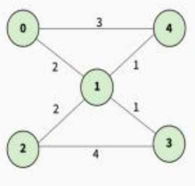
Thursday, April 17, 2025 10:11 PM

Minimum Weight Cycle

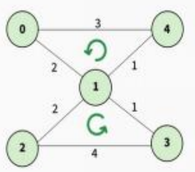
Given an **undirected**, **weighted** graph with **V** vertices numbered from 0 to V-1 and **E** edges, represented by a 2d array **edges[][]**, where **edges[i] = [u, v, w]** represents the edge between the nodes **u** and **v** having **w** edge weight.
Your task is to find the **minimum weight cycle** in this graph.

Examples:

Input: V = 5, edges[][] = [[0, 1, 2], [1, 2, 2], [1, 3, 1], [1, 4, 1], [0, 4, 3], [2, 3, 4]]

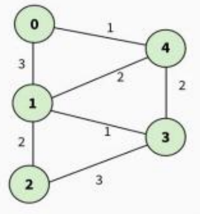


Output: 6
Explanation:

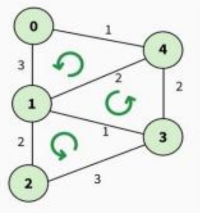


Minimum-weighted cycle is 0 → 1 → 4 → 0 with a total weight of 6(2 + 1 + 3)

Input: V = 5, edges[][] = [[0, 1, 3], [1, 2, 2], [0, 4, 1], [1, 4, 2], [1, 3, 1], [3, 4, 2], [2, 3, 3]]



Output: 5
Explanation:



Minimum-weighted cycle is 1 → 3 → 4 → 1 with a total weight of 5(1 + 2 + 2)

Constraints:
$1 \le V \le 100$
$1 \le E = edges.size() \le 10^3$
$1 \le edges[i][j] \le 100$

eg. I/p: V=5, edges[][]= [[0,1,2], [1,2,2], [1,3,1], [1,4,1], [0,4,3], [2,3,4]]

[A cycle in a graph is a path that starts & ends at the same vertex without repeating any edges or vertices [except the starting/ending vertex]. The minimum weight cycle is the one among all possible cycles that has the smallest total sum of edge weights.

[Naive Approach] Find all cycle weight :-

Find all cycles in the graph using DFS, and while exploring each cycle, keep track of its total weight. Update and maintain the minimum weight found across all such cycles

```
#include <vector>
using namespace std;
// Construct the adjacency list
test.cpp 1 of 43 problems
tor is not a template C/C++(864)
vector<vector<vector<int>>> constructAdj(int V, vector<vector<int>> &edges)
{
    vector<vector<vector<int>>> adj(V);
    for (auto &edge : edges)
    {
        int u = edge[0], v = edge[1], w = edge[2];
        adj[u].push_back({v, w});
        adj[v].push_back({u, w});
    }
    return adj;
}


int minCycle;

//DFS to explore cycles and track their weights
void dfs(int u, int parent, vector<vector<vector<int>>> &adj, vector<bool> &visited,
vector<int> &path, vector<int> &weights, int currWeight
){
    visited[u]=true;
    path.push_back(u);

    for(auto &edge: adj[u]){
        int v=edge[0];
        int w=edge[1];
        // avoid goind back to the parent
        if(v==parent)continue;
        if(!visited[v]){
            weights.push_back(w);
            dfs(v, u, adj, visited, path, weights, currWeight+w);
            weights.pop_back();
        }
        else{
            // Found a cycle
            auto it = find(path.begin(), path.int(), v);
            if(it!=path.end()){
                int cycleWeight=0;
                int idx=it-path.begin();
                for(int i=idx;i<weights.size();i++){
                    cycleWeight+=weights[i];
                }
                // add the closing edge
                cycleWeight+=w;
                minCycle=min(minCycle, cycleWeight);
            }
        }
    }
    path.pop_back();
    visited[u] = false;
}

int findMinCycle(int V, vector<vector<int>> &edges){
    vector<vector<vector<int>>> adj=constructAdj(V, edges);
    minCycle=INT_MAX;
    vector<bool> visited(V, false);
    vector<int> path, weights;
    for(int i=0;i<V;i++){
        dfs(i, -1, adj, visited, path, weights, 0);
    }
    return minCycle;
}
```

Time Complexity: O(2^V)The time complexity is exponential in the worst case (e.g. **O(2^V)**) due to the large number of simple cycles, but in typical cases, especially for sparse graphs, it may behave closer to **O(V×(V + E))**

Space Complexity: O(V+E), as dominated by the storage for the graph (adjacency list) and the DFS auxiliary structures (visited array, path, weights, and recursion stack).

[Expected Approach] : Using Djikstra's Algorithm

– O(E* (V+E)log V) Time & O(V+E) Space.

To find the shortest cycle in the graph, we iterate over edge (u,v,w) and temporarily remove it. The idea is that any edge might be part of the minimum weight cycle, so we can check

We then use Djikstra's algorithm (or any shortest path algorithm) to find the shortest path from u to v while excluding the removed edge. If such a path exits, adding the edge back completes a cycle. The total weight of this cycle is the sum of the shortest path and the weight of the removed edge.

By repeating this process for every edge, we ensure that all possible cycles are considered and we avoid redundant checks. Among all valid cycles found, we track & return the one with the minimum total weight. This method is both efficient & systematic for Identifying the shortest cycle in a weighted, directed graph.

the shortest cycle in a weighted, directed graph.

Step by step Implementation:

- First, construct the adjacency list representation of the graph based on the provided edges.
- Iterate through each edge in the graph and temporarily exclude it during the calculations.
- For each excluded edge, use Dijkstra's algorithm to compute the shortest path between the source and destination nodes.
- After calculating the shortest distance, if it's not infinity, it means a path exists between the source and destination even without the excluded edge, forming a potential cycle.
- Repeat the process for all edges and track the minimum cycle weight by adding the excluded edge's weight to the shortest path found

```
class Solution {
    private:
    // Construct adjacency list
vector<vector<vector<int>>> constructadj(int V,
                                    vector<vector<int>>& edges) {

    vector<vector<vector<int>>> adj(V);
    for (auto& edge : edges) {
        int u = edge[0], v = edge[1], w = edge[2];
        adj[u].push_back({v, w});
        adj[v].push_back({u, w});
    }
    return adj;
}
    // find shortest path between src and dest
    int shortestPath(int V, vector<vector<vector<int>>> &adj, int src, int dest){
        vector<int> dist(V, INT_MAX);
        dist[src]=0;
        //priority queue
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
        pq.push({0, src});
        while(!pq.empty()){
            pair<int, int> top=pq.top();
            pq.pop();
            int d=top.first, u=top.second;

            for(auto& neighbor: adj[u]){
                int v=neighbor[0];
                int w=neighbor[1];

                // skip the ignored case
                if((u==src && v==dest)||(u==dest && v==src))continue;

                if(dist[v]>dist[u]+w){
                    dist[v]=dist[u]+w;
                    pq.push({dist[v], v});
                }
            }
        }
        return dist[dest];
    }

  public:
    int findMinCycle(int V, vector<vector<int>>& edges) {
        // code here
        vector<vector<vector<int>>> adj=constructadj(V, edges);
        int minCycle=INT_MAX;
        for(const auto& edge: edges){
            int u = edge[0];
            int v = edge[1];
            int w = edge[2];

            int dist = shortestPath(V, adj, u, v);

            if(dist!=INT_MAX){
                minCycle = min(minCycle, dist+w);
            }
        }
        return minCycle;
    }
};
```

Time Complexity: O(E * (V + E) log V) for iterating over each edge and running Dijkstra's algorithm, which involves creating a new adjacency list and recalculating shortest paths multiple times.
Space Complexity: O(V + E) for the adjacency list, temporary edge storage, and Dijkstra's algorithm data structures like the distance array and priority queue.