Assignments week 7

Son Cao

2025-10-16

Assignment 1 - Topological sorting

My code:

```
#include <iostream>
#include "utils.h"   // for reading vectors
#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <string>
#include <algorithm> //std::min


std::unordered_map<std::string, std::vector<std::pair<std::string, int>>> graph;
std::unordered_map<std::string, int> state;
std::vector<std::string> topo;
bool has_cycle = false;

void dfs(const std::string& u){
    state[u] = 1;
    for(auto &[v, w] : graph[u]){   //v: dest node(), w: edge weight(int)
        if(state[v] == 0) dfs(v);
        else if(state[v] == 1) has_cycle = true;
    }
    state[u] = 2;
    topo.push_back(u);  //C->B->A
}

int main() {
    /* TODO:
        Write a program that reads a list of edges representing a *weighted directed* graph
        The graph is given as a comma-separated list between square brackets, e.g. `[(A, B,
        Use the class `Edge<T>` defined in `utils.h` to represent the edges of the graph.
        The program must check if the graph contains a cycle, and if it does not it must co
        It must do so by topologically sorting the nodes of the graph and then using this o
    */
    std::vector<sax::edge<std::string>> edges;  //src, dest, weight(int)
    std::cin >> edges;
    std::string start, end;
    std::cin >> start >> end;

    std::unordered_set<std::string> all_nodes;
    for(auto &edge : edges){
        graph[edge.src].push_back({edge.dest, edge.weight});
        all_nodes.insert(edge.src);
        all_nodes.insert(edge.dest);
    }
    for(auto &node : all_nodes){
        if(state[node] == 0){
            dfs(node);
        }
    }
    if(has_cycle){
        std::cout << "CYCLE\n";
        return 0;
    }
    std::reverse(topo.begin(), topo.end()); //=> A->B->C

    const int INF = 1e9;  //infinity
    std::unordered_map<std::string, int> dist;  //initially dont know the distance so set t
    for(auto &node : all_nodes) dist[node] = INF;
        dist[start] = 0;  //base case

    for(auto &u : topo){
        if(dist[u] != INF){
            for(auto &[v, w] : graph[u]){
                dist[v] = std::min(dist[v], dist[u] + w);
            }
        }
    }
    if(dist[end] == INF){
        std::cout << "NO PATH\n";
    } else {
        std::cout << dist[end] << "\n";
    }
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n + m), because the algorithm visits each node once in the DFS for topological sorting. Then, it relaxes each edge once in the shortest-path phase.

Assignment 2 - Fire department placement

My code:

```
#include <iostream>
#include "utils.h"   // for reading vectors
#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <string>
#include <queue>
#include <algorithm>

int main() {
    /* TODO:
        A municipality wants to place a fire department in one of its neighborhoods such tha
        from the fire department to all other neighborhoods is minimized.

        Write a program that computes its optimal placement and prints the smallest sum of d
        department to all other neighborhoods.

        The program must read a list of edges representing an *undirected* graph from its st
        comma-separated list between square brackets, e.g. `[(A, B, 5),(B, C, 10),(C, D, 2)]
        indicates that there is a road between neighborhoods A and B with length w.

        The program must then compute at which neighborhood the fire department should be pl
        as well as the sum of the distances from that neighborhood to all other neighborhoo

        The time complexity of your solution must be `O(n m log n)`, where `n` is the numbe
        of edges in the graph.
    */
    std::vector<sax::edge<std::string>> edges;
    std::cin >> edges;

    std::unordered_map<std::string, std::vector<std::pair<std::string, int>>> graph;
    std::unordered_set<std::string> all_nodes;

    for(auto& edge : edges){
        graph[edge.src].push_back({edge.dest, edge.weight});
        graph[edge.dest].push_back({edge.src, edge.weight});
        all_nodes.insert(edge.src);
        all_nodes.insert(edge.dest);
    }

    const int INF = 1e9;
    std::string best_node;
    int best_sum = INF;

    //dijkstra from each node
    for(auto& start : all_nodes){
        std::unordered_map<std::string, int> dist;
        for(auto &node : all_nodes) dist[node] = INF;
        dist[start] = 0;

        //in Dijkstra's algorithm, we want the smallest distance (the "closest" node first)
        //So we need to flip the order to make it a min-heap - and that's what std::greater
        std::priority_queue<std::pair<int, std::string>, std::vector<std::pair<int, std::str
        pq.push({0, start});

        while(!pq.empty()){
            auto [d, u] = pq.top();  //u: current node being visited  //top() gives node wi
            pq.pop();                //d: current distance from start to u  //pop() the sma
            if(d > dist[u]) continue;  //we maay input shorter path to the same node later.

            for(auto &[v, w] : graph[u]){
                if(dist[v] > dist[u] + w){
                    dist[v] = dist[u] + w;
                    pq.push({dist[v], v});
                }
            }
        }
        int total = 0;
        for(auto &[n, d] : dist){
            if(d != INF) total += d;
        }

        if(total < best_sum){
            best_sum = total;
            best_node = start;
        }
    }
    std::cout << best_node << ", " << best_sum << "\n";
    return 0;
}

// Start     To A     To B     To C     Total
// A          0        2        5         7
// B          2        0        3         5
// C         5       3       0        8

// Start = A => total = 7
// => best_sum = 7, best_node = "A"

// Start = B → total = 5
// => 5 < 7 => update: best_sum = 5, best_node = "B"

// Start = C => total = 8
// => 8 > 5 => no update
```

Time complexity: this algorithm has a time complexity of O(n * m log n), because the program runs Dijkstra's algorithm once for every node in the graph. One Dijkstra run takes O(m log n) time using a priority queue (std::priority_queue). Since we repeat it for each node (n times) => total = O(n * m log n).

Assignment 3 - Undirected cycle detection

My code:

```
#include <iostream>
#include <unordered_map>
#include "utils.h"
#include <vector>
#include <string>

//DSU structure
std::unordered_map<std::string, std::string> parent;
std::unordered_map<std::string, int> size;

//find function with path compression
std::string find(const std::string &x){
    if(parent[x] == x) return x;
    return parent[x] = find(parent[x]); //path compression
}

bool union_set(const std::string &a, const std::string &b){
    std::string rootA = find(a);
    std::string rootB = find(b);
    if(rootA == rootB) return false; //cuz cycle detected  //if both node have the same roo
                                     //so adding edge between them would make cycle
    if(size[rootA] < size[rootB]){
        std::swap(rootA, rootB);
    }
    parent[rootB] = rootA;
    size[rootA] += size[rootB];
    return true;
}


int main() {
    /* TODO:
        Write a program that reads a list of edges representing an *undirected* graph from
        input (given as a comma-separated list between square brackets, e.g. `[(A, B),(B, C

        The program must add these edges, one by one, to an initially empty graph, and befo
        edge, it must determine whether that edge would introduce a cycle into the graph.

        In case the edge would introduce a cycle, the edge must not be added to the graph a
        should stop processing further edges. Otherwise, the edge is added to the graph and
        continues with the next edge.

        The program must then print how many edges were added before the first cycle was cr

        Your program must run in `O(m log(n))` time, where `n` is the number of nodes and `
        edges.
    */
   //didnt use sax::edge cuz we dont need weights
    std::vector<std::pair<std::string, std::string>> edges;
    std::cin >> edges;
    int added = 0;

    for(auto &edge : edges){
        //initialize parent and size if new node
        if(!parent.count(edge.first)){
            parent[edge.first] = edge.first;
            size[edge.first] = 1;
        }
        if(!parent.count(edge.second)){
            parent[edge.second] = edge.second;
            size[edge.second] = 1;
        }
        if(!union_set(edge.first, edge.second)){
            //cycle detected
            break;
        }
        added++;
    }
    std::cout << added << "\n";
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(m log n), because each edge is processed once => m times total. For each edge, the algorithm performs find() and union() operations on the disjoint set. With path compression and union by size, each operation runs in almost constant time => technically O(log n) (or even tighter, O(alpha(n)), where alpha is the inverse Ackermann function).

Assignment 4 - Minimum satellite communication network

My code:

```cpp
#include <iostream>
#include "utils.h"
#include "coordinate.h"
#include <unordered_map>
#include <cmath>
#include <algorithm>
#include <vector>
#include <edge.h>

// Euclidean Distance
// For two coordinates (x1, y1, z1) and (x2, y2, z2):
// sqrt((x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2)

// Kruskal's algorithm steps:
    // Compute all possible edges (for every pair of satellites).
    // Sort edges by weight (distance).
    // Initialize DSU (each satellite is its own parent).
    // For each edge (smallest first):
    // If connecting them doesn't form a cycle (=>union_set() returns true), add its weight
    // Stop when you have added n - 1 edges.


//cannot use sax::edge cuz it's always int weight
struct Edge{
    int src, dest;
    double weight;
};

// We use std::vector<int> parent, size; instead of unordered_map because:
    // Satellite nodes are numbered 0 to n - 1, not arbitrary names.
    // Vectors allow direct fast access (O(1)), while maps need hash lookups (slower).
    // We can initialize all nodes easily with a simple loop.
// std::unordered_map<int, int> parent;
// std::unordered_map<int, int> size;
std::vector<int> parent, size;

int find(int x){
    if(parent[x] == x) return x;
    return parent[x] = find(parent[x]);  //path compression
}

bool union_set(int a, int b){
    int rootA = find(a);
    int rootB = find(b);
    if(rootA == rootB) return false; //cycle detected

    if(size[rootA] < size[rootB]){
        std::swap(rootA, rootB);
    }
    parent[rootB] = rootA;
    size[rootA] += size[rootB];
    return true;
}

// //compute Euclidean distance between two coordinates
// double distance(const sax::coordinate &a, const sax::coordinate &b){
//     double dx = a.x - b.x;
//     double dy = a.y - b.y;
//     double dz = a.z - b.z;
//     return std::sqrt(dx*dx + dy*dy + dz*dz);
// }

int main()
{
    /* TODO:
        A communication network consists of a number of satellites that need to be connecte
        Satellites can be connected directly to each other, or indirectly through other sate
        how the internet works.

        Since communication is costly, the network must be designed such that the total cos
        satellites is minimized. Also, the network must not contain any cycles, since that w
        resources. In other words, there should be just one unique communication path betwe

        Given (x, y, z) coordinates of satellites, write a program that computes the minimum
        satellites in a communication network, where the cost to connect two satellites is g
        distance between them.

        The program receives a list of satellites from its standard input (given as a comma-
        square brackets, e.g. `[(0,0,0),(1,1,1),(2,2,2)]`), where each satellite is represen
        coordinates. Use the `sax::coordinate` type to represent the coordinates of a satell

        The program must then compute the minimum cost to connect all satellites in a commun
        this cost to its standard output with two digits after the decimal point (e.g. `3.4
    */
    std::vector<sax::coordinate> satellites;
    std::cin >> satellites;

    int n = satellites.size();
    if(n <= 1){
        std::cout << "0.00\n";
        return 0;
    }

    std::vector<Edge> edges;

    for(int i = 0; i < n; i++){
        for(int j = i + 1; j < n; j++){
            edges.push_back({i, j, satellites[i].distance_to(satellites[j])});
        }
    }
    //ex: satellites[0] = (0, 0, 0)  => index = 0
    //    satellites[1] = (1, 0, 0)  => index = 1
    //    satellites[2] = (0, 1, 0)  => index = 2

    //edges.push_back({0, 1, satellites[0].distance_to(satellites[1])});
    //=> Edge created: {src=0, dest=1, weight=1.00}

    //edges.push_back({0, 2, satellites[0].distance_to(satellites[2])});
    //=> Edge created: {src=0, dest=2, weight=1.00}

    //edges.push_back({1, 2, satellites[1].distance_to(satellites[2])});
    //=>Edge created: {src=1, dest=2, weight=1.41}

    //sort edges by weight
    std::sort(edges.begin(), edges.end(), [](const Edge &a, const Edge &b){
        return a.weight < b.weight;
    });

    //using vector not unordered_map so need to change to this
    parent.resize(n);           //allocate space for all n satellites
    size.resize(n, 1);          //size is initialized to 1 for each node
    //DSU - set parent and size for later use in union_set and find
    for(int i = 0; i < n; i++){
        parent[i] = i;
        // size[i] = 1;
    }

    double total = 0.0;
    int edges_used = 0;

// Kruskal's Algorithm
    // Kruskal doesn't care about the coordinates, only about the indices (0,1,2).
    // It uses these indices in the DSU (Union-Find) to merge sets:
    // Connect 0-1
    // Connect 0-2
    // Don't connect 1-2 (would make a cycle )
    // Minimum total = 1.00 + 1.00 = 2.00

    for(auto &edge : edges){
        if(union_set(edge.src, edge.dest)){
            total += edge.weight;
            edges_used++;

            if(edges_used == n-1){
                break; //MST complete
            }
        }
    }
    std::cout.setf(std::ios::fixed);
    std::cout.precision(2);
    std::cout << total << "\n";
    return 0;
}
```

I KEEP INSPECTING ERROR IN THIS TEST (I've also asked some of my friend they got the same problem): The answer for the following test input (line 3) is incorrect: [(0,0,0),(1,2,3),(8,5,2),(2,3,0),(5,5,5)] got 16.55 expected 22.71

Time complexity: this algorithm has a time complexity of . . . .., because . . . . . .

Assignment 5 - Shortest paths on minimum spanning trees

My code:

```
#include <iostream>
#include "utils.h"
#include "coordinate.h"
#include <unordered_map>
#include <cmath>
#include <algorithm>
```

```
#include <vector>
#include <edge.h>
#include <string>

std::unordered_map<std::string, std::string> parent;
std::unordered_map<std::string, int> size;

std::string find(const std::string &x){
    if(parent[x] == x) return x;
    return parent[x] = find(parent[x]);
}

bool union_set(const std::string &a, const std::string &b){
    std::string rootA = find(a);
    std::string rootB = find(b);
    if(rootA == rootB) return false;

    if(size[rootA] < size[rootB]){
        std::swap(rootA, rootB);
    }
    parent[rootB] = rootA;
    size[rootA] += size[rootB];
    return true;
}

// In an MST or any undirected graph, there can be multiple routes between two nodes.
// The old DFS stopped after the first one it found,
// but the new DFS tries all paths and keeps the smallest cost.
void dfs(const std::string &current, const std::string &target, std::unordered_map<std::stri
        std::unordered_map<std::string, bool> &visited, int current_cost, int &min_cost){
    if(current == target){
        min_cost = std::min(min_cost, current_cost);
        return;
    }
    visited[current] = true;
    for(auto &[neighbor, w] : graph[current]){
        if(!visited[neighbor]){
            dfs(neighbor, target, graph, visited, current_cost + w, min_cost);
        }
    }
    visited[current] = false;
}

int main() {
    /* TODO:
        Given an undirected graph, write a program that computes the minimum spanning tree
        either Prim's or Kruskal's algorithm, and then computes the shortest path (i.e., wi
        cost) between two given nodes on the minimum spanning tree.

        The program receives a list of edges representing an *undirected weighted* graph fr
        input (given as a comma-separated list between square brackets, e.g. `[(A, B, 5),(B
        followed by two characters representing the start and end nodes (e.g. `A D`) - simi
        of this week.

        The program's output must be the cost of the shortest path between the start and en
        spanning tree, or `NO PATH` if there is no path between the start and end nodes.
    */
    //ex: [(A,B,5),(B,C,10),(C,D,2)] A D
    std::vector<sax::edge<std::string>> edges;
    std::cin >> edges;
    std::string start, end;
    std::cin >> start >> end;

    for(auto &edge : edges){
        if(!parent.count(edge.src)){
            parent[edge.src] = edge.src;  //set parent and size for later use in union_set
            size[edge.src] = 1;
        }
        if(!parent.count(edge.dest)){
            parent[edge.dest] = edge.dest;
            size[edge.dest] = 1;
        }
    }
    //Kruskal's algorithm -> build MST
    std::sort(edges.begin(), edges.end(), [](auto &a, auto &b){
        if (a.weight != b.weight) return a.weight < b.weight;
        if (a.src != b.src) return a.src < b.src;
        return a.dest < b.dest;
    });

    std::unordered_map<std::string, std::vector<std::pair<std::string, int>>> graph;
    for(auto &edge : edges){
        if(union_set(edge.src, edge.dest)){  //build graph after union_set to avoid cycle
            graph[edge.src].push_back({edge.dest, edge.weight});
            graph[edge.dest].push_back({edge.src, edge.weight});
        }
    }
    std::unordered_map<std::string, bool> visited;
    const int INF = 1e9;
    int min_cost = INF;

    dfs(start, end, graph, visited, 0, min_cost);
```

```
    if (min_cost != INF){
        std::cout << min_cost << std::endl;
    }
    else{
        std::cout << "NO PATH" << std::endl;
    }
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(m log n), because when building the MST (Kruskal's algorithm): Sorting all m edges takes O(m log m), which is roughly O(m log n) since m <= n^2. Each union_set and find operation takes almost constant time (O(log n)) due to path compression and union by size. So total for MST = O(m log n). while the DFS traversal of the MST only adds linear time => the MST has exactly n - 1 edges => so DFS runs in O(n + (n - 1)) = O(n) time.