Assignments week 6

Ethan Bastian, Abdullah Turk

2025-09-05

Assignment YY - the quick brown fox jumps over the lazy dog

My code:

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

// BFS that returns the path from start to target as a vector of chars
std::vector<char> bfs_path(const std::unordered_map<char, std::vector<char>>& adj, char sta
    if (adj.find(start) == adj.end() || adj.find(target) == adj.end()) return {};

    std::queue<char> q;
    std::unordered_set<char> visited;
    std::unordered_map<char, char> parent;

    visited.insert(start);
    q.push(start);

    bool found = false;
    while (!q.empty() && !found) {
        char cur = q.front(); q.pop();
        auto it = adj.find(cur);
        if (it == adj.end()) continue;
        for (char nei : it->second) {
            if (!visited.count(nei)) {
                visited.insert(nei);
                parent[nei] = cur;
```

```
                if (nei == target) { found = true; break; }
                q.push(nei);
            }
        }
    }

    if (!found) return {};

    // reconstruct path
    std::vector<char> path;
    for (char v = target; ; v = parent[v]) {
        path.push_back(v);
        if (v == start) break;
    }
    std::reverse(path.begin(), path.end());
    return path;
}

int main() {
    // Read edges as pairs of char, then start and end as char
    std::vector<std::pair<char, char>> edges;
    char start, end;
    if (!(std::cin >> edges >> start >> end)) {
        std::cerr << "Error reading input" << std::endl;
        return 1;
    }

    std::unordered_map<char, std::vector<char>> graph;
    for (const auto& e : edges) {
        graph[e.first].push_back(e.second);
        graph[e.second].push_back(e.first); // undirected
    }

    auto path = bfs_path(graph, start, end);
    if (path.empty()) {
        std::cout << "[]";
        return 0;
    }

    // print as list of edges: [(A, B), (B, C)]
    std::cout << "[";
    for (size_t i = 0; i + 1 < path.size(); ++i) {
        std::cout << "(" << path[i] << ", " << path[i+1] << ")";
        if (i + 2 < path.size()) std::cout << ", ";
    }
    std::cout << "]";
```

```
    return 0;
}
```

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

Assignment ZZ - the quick brown fox jumps over the lazy dog

My code:

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

int main() {
    /* TODO:
        Write a program that reads a list of edges representing an *undirected* graph
        from its standard input (given as a comma-separated list between square
        brackets, e.g. `[(A, B),(B, C),(C, D)]`), followed by two characters representing
        the start and end nodes.

        The program must then find the path with the fewest edges between the start and
        end nodes, and print the number of edges of that path, or `-1` if no such path exis
    */

    std::vector<std::pair<char, char>> edges;
    char start;
    char end;

    std::cin >> edges >> start >> end;

    std::unordered_map<char, std::vector<char>> graph;

    for (auto& e: edges) {
        graph[e.first].push_back(e.second);
        graph[e.second].push_back(e.first);
    }

    std::queue<std::pair<char, int>> q;
    std::unordered_set<char> visited;
```

```
    q.push({start, 0});
    visited.insert(start);

    while (!q.empty()) {
        auto [node, dist] = q.front();
        q.pop();
        if (node == end) {
            std::cout << dist << std::endl;
            return 0;
        }

        for (auto& neighbor : graph[node]) {
            if (visited.find(neighbor) == visited.end()) {
                visited.insert(neighbor);
                q.push({neighbor, dist + 1});
            }
        }
    }
    std::cout << -1 << std::endl;


    return 0;
}
```

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

Assignment AA - brace yourselves, winter is coming

My code:

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

int main() {
    std::vector<std::pair<char, char>> edges;
    std::cin >> edges;

    std::unordered_map<char, std::vector<char>> adj;
```

```
    std::unordered_map<char, int> in_degree;
    std::unordered_set<char> nodes;

    // Build graph and calculate in-degrees
    for (const auto& [u, v] : edges) {
        adj[u].push_back(v);
        nodes.insert(u);
        nodes.insert(v);
        in_degree[v]++;
        if (in_degree.find(u) == in_degree.end()) {
            in_degree[u] = 0;
        }
    }

    // Kahn's algorithm with min-heap for lexicographic order
    std::priority_queue<char, std::vector<char>, std::greater<char>> q;
    for (const auto& node : nodes) {
        if (in_degree[node] == 0) {
            q.push(node);
        }
    }

    std::vector<char> result;
    while (!q.empty()) {
        char u = q.top();
        q.pop();
        result.push_back(u);

        for (char v : adj[u]) {
            in_degree[v]--;
            if (in_degree[v] == 0) {
                q.push(v);
            }
        }
    }

    if (result.size() != nodes.size()) {
        //std::cout << "hey pal, there's a big old cycle there" << std::endl;
        std::cout << "CYCLE" << std::endl;
    } else {
        std::cout << result << std::endl;
    }

    return 0;
}
```

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

Assignment AA - brace yourselves, winter is coming

My code:

```cpp
#include <iostream>
#include <queue>
#include <unordered_map>
#include <unordered_set>
#include <utility>
#include <vector>
#include "utils.h"

int main()
{
    /* TODO:
        Write a program that reads a list of edges representing an *undirected* graph
        from its standard input (given as a comma-separated list between square brackets).
        The graph may be disconnected, meaning that there may be multiple connected componen

        The program must then compute and print the size of the largest connected component
        in the graph.

        The time complexity of your solution must be `O(n + m)`, where `n` is the number of
        and `m` is the number of edges in the graph.
    */

    std::vector<std::pair<int, int>> edges;
    std::cin >> edges;

    std::unordered_map<int, std::vector<int>> graph;
    std::unordered_set<int> nodes;

    for (auto& e: edges) {
        graph[e.first].push_back(e.second);
        graph[e.second].push_back(e.first);

        nodes.insert(e.first);
        nodes.insert(e.second);

    }
```

```
    std::unordered_set<int> visited;
    int largest = 0;

    for (int node : nodes) {
        if (visited.find(node) != visited.end()) continue;
        int size = 0;

        std::queue<int> q;
        q.push(node);
        visited.insert(node);

        while (!q.empty()) {
            int current = q.front();
            q.pop();
            size++;

            for(int neighbor : graph[current]) {
                if (visited.find(neighbor) == visited.end()) {
                    visited.insert(neighbor);
                    q.push(neighbor);
                }
            }
        }

        if (size > largest) largest = size;
    }

    std::cout << largest << std::endl;

    return 0;
}
```

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

Assignment AA - brace yourselves, winter is coming

My code:

```
#include <iostream>
#include <unordered_map>
#include <unordered_set>
#include <vector>
```

```cpp
#include <algorithm>
#include "utils.h"

bool dfs(int node,
         const std::unordered_map<int, std::vector<int>>& adj,
         std::unordered_set<int>& visited,
         std::unordered_set<int>& rec_stack,
         std::vector<int>& path,
         std::vector<std::pair<int, int>>& cycle) {

    visited.insert(node);
    rec_stack.insert(node);
    path.push_back(node);

    if (adj.find(node) != adj.end()) {
        for (int neighbor : adj.at(node)) {
            if (rec_stack.count(neighbor)) {
                // Found cycle
                auto it = std::find(path.begin(), path.end(), neighbor);
                for (auto curr = it; curr != path.end() - 1; curr++) {
                    cycle.push_back({*curr, *(curr + 1)});
                }
                cycle.push_back({path.back(), neighbor});
                return true;
            }
            if (!visited.count(neighbor)) {
                if (dfs(neighbor, adj, visited, rec_stack, path, cycle)) {
                    return true;
                }
            }
        }
    }

    rec_stack.erase(node);
    path.pop_back();
    return false;
}

int main() {
    std::vector<std::pair<int, int>> edges;
    std::cin >> edges;

    std::unordered_map<int, std::vector<int>> adj;
    std::unordered_set<int> nodes;

    for (const auto& [u, v] : edges) {
```

```
        adj[u].push_back(v);
        nodes.insert(u);
        nodes.insert(v);
    }

    std::unordered_set<int> visited;
    std::unordered_set<int> rec_stack;
    std::vector<std::pair<int, int>> cycle;
    std::vector<int> path;

    for (int node : nodes) {
        if (!visited.count(node)) {
            if (dfs(node, adj, visited, rec_stack, path, cycle)) {
                std::cout << cycle << std::endl;
                return 0;
            }
        }
    }

    std::cout << "[]" << std::endl;
    return 0;
}
```

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

Assignment AA - brace yourselves, winter is coming

My code:

```
#include "utils.h"
#include <iostream>
#include <queue>
#include <unordered_map>
#include <unordered_set>
#include <utility>
#include <vector>



int count_paths(int current, int end, std::unordered_map<int, std::vector<int>>& graph, std
    if (current == end) return 1;

    visited.insert(current);
```

```
    int total = 0;

    for (int neighbor : graph[current]) {
        if (visited.find(neighbor) == visited.end()) {
            total += count_paths(neighbor, end, graph, visited);
        }
    }

    visited.erase(current);
    return total;
}




int main() {
  /* TODO:
      Write a program that reads a list of edges representing an *undirected*
     graph from its standard input (given as a comma-separated list between
     square brackets), followed by two integers representing the start and end
     nodes.

      The program must then compute`` and print the number of *different*
     "simple" paths from the start node to the end node. A simple path is a path
     that does not visit any node more than once - in other words, cycles are
     not allowed.
  */

    std::vector<std::pair<int, int>> edges;
    int start, end;

    // Example input: [(0,1),(1,2),(2,0)] 0 2
    std::cin >> edges >> start >> end;

    std::unordered_map<int, std::vector<int>> graph;
    for (auto& e : edges) {
        graph[e.first].push_back(e.second);
        graph[e.second].push_back(e.first);
    }

    std::unordered_set<int> visited;
    int result = count_paths(start, end, graph, visited);

    std::cout << result << std::endl;
    return 0;
```

```
    return 0;
}
```

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