Assignments week 6: Graph

Son Cao

2025-09-08

Assignment 1 - Pathfinding

My code:

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


//std::unordered_map<char, std::vector<char>>& graph: this is the adjacency list of the grap
//unordered_map<char, vector<char>> => Maps each node (char) to its neighbors (vector<char>)

//std::unordered_map<char, char>& parent: this keeps track of how each node was reached so u
bool bfs(char start, char end, const std::unordered_map<char, std::vector<char>>& graph, std
    std::unordered_set<char> visited;
    //unordered_set<char> => Stores unique characters (char) in no particular order
    std::queue<char> q;

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

    while(!q.empty()){
        char node = q.front();
        q.pop();  //pop always remove the front element of the queue

        if(node == end){
            return true; //found the end node
        }
```

```
        for(char neighbor : graph.at(node)){
            if(!visited.contains(neighbor)){
                visited.insert(neighbor);
                parent[neighbor] = node;  //track how we reached this neighbor ex: neigbor =
                q.push(neighbor);
            }
        }
    }
    return false;
}

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 characters representing the start and end nodes.

        The program must then find out whether there exists a path from the start node to
        the end node. If such a path exists, the program must print the path as a
        comma-separated list of edges between square brackets (e.g. `[(A, B), (B, C)]`),
        otherwise it must print `[]`.
    */
   //std::pair is useful when you want to store two related values together as a single obj
    std::vector<std::pair<char, char>> edges;
    char start, end;

    //[(A, B),(B, C),(C, D)] A D
    if(!(std::cin >> edges >> start >> end)){
        std::cerr << "Error reading input" <<std::endl;
        return 1;
    }

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

    //parent map for reconstructing the path
    std::unordered_map<char, char> parent;

    bool found = bfs(start, end, graph, parent);

    if(!found){
        std::cout << "[]" << std::endl;
```

```
        return 0;
    }

    //reconstruct path (end -> start)
    std::vector<std::pair<char, char>> path;
    char current = end;
    while(current != start){
        char prev = parent[current];  //B = parent[C]
        path.push_back({prev, current}); //Add the edge (prev, current) to the path. (B, C)
        current = prev; //Move backward in the path - now current becomes the parent node. T
    }

    //reverse path so it goes from start to end
    std::reverse(path.begin(), path.end());

    std::cout << "[";
    for(size_t i = 0; i < path.size(); i++){
        std::cout << "(" << path[i].first << ", " << path[i].second << ")";
        if(i < path.size() - 1) std::cout << ", ";
    }
    std::cout << "]" << std::endl;

    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n + m), because each node and edge is processed at most once during BFS traversal.

Assignment 2 - Shortest paths

My code:

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


int bfs_shortest_path(char start, char end, const std::unordered_map<char, std::vector<char>
    std::unordered_set<char> visited;
    std::queue<std::pair<char, int>> q;  //queue of node + distance from start

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

```
    while(!q.empty()){
        auto[node, dist] = q.front();
        q.pop();

        if(node == end){
            return dist;    //found shortest path
        }

        for(char neighbor : graph.at(node)){
            if(!visited.contains(neighbor)){
                visited.insert(neighbor);
                q.push({neighbor, dist + 1}); // when going from node to neighbor need to i
            }
        }
    }
    return -1; //end not reachable

// The function finds the shortest path because BFS explores nodes level by level:
// First it visits all nodes 1 edge away from the start,
// Then all nodes 2 edges away,
// Then 3 edges away, and so on.
// the first time BFS reaches the end node, it must be via the fewest possible edges => tha

}
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, end;

   if(!(std::cin >> edges >> start >> end)){
    std::cerr << "Failed reading input" << std::endl;
    return 1;
   }

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

```
    graph[edge.second].push_back(edge.first);
   }

   int shortest_dist = bfs_shortest_path(start, end, graph);
   std::cout << shortest_dist << std::endl;
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n + m), because each node and edge is processed once while performing the breadth-first search.

Assignment 3 - Ordering dependencies

My code:

```
#include <iostream>
#include "utils.h"
#include <unordered_map>
#include <unordered_set>
#include <vector>
#include <stack>
#include <algorithm> //std::reverse

// DFS-based topological sort:
    // Start at an unvisited node.
    // Recursively visit all its neighbors (dependencies).
    // When finished visiting all neighbors, add the node to the ordering.
    // Use visited/visiting states to detect cycles.

//0: unvisited
//1: visiting
//2: visited
bool dfs(char node, const std::unordered_map<char, std::vector<char>>& graph, std::unordered
    state[node] = 1; //visiting

    for(char neighbor : graph.at(node)){
        if(state[neighbor] == 1) return true; //cycle detected
        if(state[neighbor] == 0){
            if(dfs(neighbor, graph, state, ordering)) return true;
        }
    }
    state[node] = 2; //visited
    ordering.push_back(node);
    return false;
}
```

```cpp
int main() {
    /* TODO:
        Write a program that reads a list of dependencies from its standard input (given
        as a comma-separated list between square brackets, e.g. `[(A,B),(B,C)]`), where
        each dependency `(A,B)` indicates that task A must be completed before task B
        can be started.

        The program must then compute and print a valid order in which the tasks can be
        completed, or `CYCLE` if no such order exists (i.e., if there is a cycle in the
        dependencies).

        If a valid order exists, it must be printed as a comma-separated list between
        square brackets (e.g. `[A,B,C]`).
    */
    std::vector<std::pair<char, char>> edges;
    if(!(std::cin >> edges)){
        std::cerr << "Failed reading input" << std::endl;
        return 1;
    }

    std::unordered_map<char, std::vector<char>> graph;
    std::unordered_set<char> all_nodes;
    for(const auto& edge : edges){
        graph[edge.first].push_back(edge.second);
        graph[edge.second]; //ensure node exist
        all_nodes.insert(edge.first);
        all_nodes.insert(edge.second);
    }


    std::unordered_map<char, int> state;
    std::vector<char> ordering;
    for(char node : all_nodes){
        state[node] = 0;
    }
    //dfs
    for(char node : all_nodes){
        if(state[node] == 0){
            if(dfs(node, graph, state, ordering)){
                std::cout << "CYCLE" << std::endl;
                return 0;
            }
        }
    }
    //[C, B, A] => [A, B, C]
    //reverse to get correct topological order
```

```
    std::reverse(ordering.begin(), ordering.end());

    std::cout << "[";
    for(size_t i = 0; i < ordering.size(); i++){
        std::cout << ordering[i];
        if(i < ordering.size() - 1) std::cout << ", ";
    }
    std::cout << "]" << std::endl;
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n + m), because each node (n) and each edge (m) in the graph is visited exactly once during the depth-first search. The algorithm only performs constant-time operations per node and edge (marking states, checking neighbors, adding to the ordering), so the total work grows linearly with the size of the graph.

Assignment 4 - Largest component

My code:

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

int dfs(char node, std::unordered_map<int, std::vector<int>>& graph, std::unordered_set<int>
    visited.insert(node);
    int size = 1;
    for(char neighbor : graph.at(node)){
        if(!visited.contains(neighbor)){
            size += dfs(neighbor, graph, visited);
        }
    }
    return size;
}

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;
    if(!(std::cin >> edges)){
        std::cerr << "Failed reading input" << std::endl;
        return 1;
    }

    std::unordered_map<int, std::vector<int>> graph;
    std::unordered_set<int> all_nodes;
    for(const auto& edge : edges){
        graph[edge.first].push_back(edge.second);
        graph[edge.second].push_back(edge.first);  //undirected graph
        all_nodes.insert(edge.first);
        all_nodes.insert(edge.second);
    }

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

    for(int node : all_nodes){
        if(!visited.contains(node)){
            int size = dfs(node, graph, visited);
            largest = std::max(largest, size);
        }
    }

    std::cout << largest << std::endl;
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n + m), because each node (n) and each edge (m) is visited exactly once during the depth-first search (DFS). Each node is marked as visited the first time it is reached, and every edge is explored at most twice (once from each endpoint), so the total work grows linearly with the size of the graph.

Assignment 5 - Cycle detection

My code:

```cpp
#include <iostream>
#include <unordered_map>
#include <algorithm> // std::find
#include "utils.h"
#include <unordered_set>

bool dfs(int node, std::unordered_map<int, std::vector<int>>& graph,
    std::unordered_map<int, int>& state,
    std::unordered_map<int, int>& parent,   //track parent tp reconstruct cycle
    int& cycle_start, int& cycle_end){

    state[node] = 1; //1: visiting

    for(char neighbor : graph.at(node)){
        if(state[neighbor] == 0){
            parent[neighbor] = node;
            if(dfs(neighbor, graph, state, parent, cycle_start, cycle_end)) return true;
        } else if(state[neighbor] == 1){  //cycle detected
            cycle_start = neighbor;    //1 is where the cycle starts (the node we loop back
            cycle_end = node;   //3 is where the cycle ends (the node that creates the back
            return true;
        }
    }
    //if visited
    state[node] = 2;
    return false;
}

int main() {
    /* TODO:
        Write a program that reads a list of edges representing a *directed* graph
        from its standard input (given as a comma-separated list between square
        brackets, e.g. `[(0,1),(1,2),(2,0)]` - see assignment 1).

        The program must then determine whether the graph contains a cycle.
        If a cycle exists, the program must print the edges in the cycle as a
        comma-separated list between square brackets (e.g. `[(0,1),(1,2),(2,0)]`),
        otherwise it must print `[]`.
    */
    std::vector<std::pair<int,int>> edges;
    if(!(std::cin >> edges)){
        std::cerr << "Failed reading input\n";
        return 1;
   }

    std::unordered_map<int, std::vector<int>> graph;
```

```
std::unordered_set<int> all_nodes;
for(const auto& edge : edges){
    graph[edge.first].push_back(edge.second);
    graph[edge.second]; //ensure node exist in the map
    all_nodes.insert(edge.first);
    all_nodes.insert(edge.second);
}

std::unordered_map<int, int> state;
std::unordered_map<int, int> parent;
for(int node : all_nodes){
    state[node] = 0;
}

int cycle_start = -1, cycle_end = -1;
bool found = false;

for(int node : all_nodes){
    if(state[node] == 0){
        if(dfs(node, graph, state, parent, cycle_start, cycle_end)){
            found = true;
            break;
        }
    }
}
if(!found){
    std::cout << "[]" << std::endl;
    return 0;
}

//reconstruct cycle edges
std::vector<std::pair<int,int>> cycle_edges;
int current = cycle_end;
cycle_edges.push_back({current, cycle_start}); //Ex: (3, 1)  //adds the edge that closes
while(current != cycle_start){
    int prev = parent[current];
    cycle_edges.push_back({prev, current});    //ex: (2,3), (1,2)
    current = prev;
}
//OR PUT THIS LINE AFTER THE LOOP FOR THE CLOSING EDGE
//cycle_edges.push_back({cycle_end, cycle_start});

std::reverse(cycle_edges.begin(), cycle_edges.end()); //then reverse => [(1,2), (2,3),

std::cout << "[";
for(size_t i = 0; i < cycle_edges.size(); i++){
```

```
        std::cout << "(" << cycle_edges[i].first << ", " << cycle_edges[i].second << ")";
        if(i != cycle_edges.size() - 1) std::cout << ", ";
    }
    std::cout << "]" << std::endl;
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n + m), because each node (n) and each edge (m) in the directed graph is visited exactly once during the DFS traversal. The cycle detection check (state[neighbor] == 1) and parent tracking both happen in constant time for each edge, so the total work done grows linearly with the number of nodes and edges.

Assignment 6 - Counting paths

My code:

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

int CountPath(int current, int end, std::unordered_map<int, std::vector<int>>& graph, std::u
    if(current == end) return 1; //path found

    visited.insert(current);
    int total = 0;

    for(int neighbor : graph[current]){
        if(!visited.count(neighbor)){   //contain return true false, count return 1, 0
            total += CountPath(neighbor, end, graph, visited);
        }
    }
    visited.erase(current);
    return total;
}

// 4 <- 1 -> 2 -> 3
//2 path
//1 -> 2 -> 3
//1 -> 4 -> 2 -> 3



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;
    if(!(std::cin >> edges >> start >> end)){
        std::cerr << "Failed reading input" << std::endl;
        return 1;
   }
    std::unordered_map<int, std::vector<int>> graph;
    for(auto& edge : edges){
        graph[edge.first].push_back(edge.second);
        graph[edge.second].push_back(edge.first); // undirected graph
    }
    std::unordered_set<int> visited;
    int total_paths = CountPath(start, end, graph, visited);

    std::cout << total_paths << std::endl;

    return 0;
}

//EX:
//Edges: [(0,1),(0,2),(1,2),(2,3)]
//Start: 0
//End: 3
    //0 -> 1 -> 2 -> 3 (YES)
    //0 -> 2 -> 3 (YES)
    //0 -> 2 -> 1 -> 2 -> 3 (NO) (not simple, revisits 2)
```

Time complexity: this algorithm has a time complexity of O(k × (n + m)), but effectively exponential in the worst case (often written as O(2^n)), because it explores all possible simple paths between the start and end nodes using recursive DFS backtracking. At each node, the algorithm can choose to visit any unvisited neighbor, and in dense graphs, this branching grows exponentially. The visited set ensures no node is revisited within a single path, but the algorithm still explores every possible combination of paths, which makes it exponential even though each DFS traversal itself touches nodes and edges in O(n + m) time.