#### Code [ SFML ]

#### Header files

#pragma once

#include <SFML/Graphics.hpp>

#include<SFML/Window.hpp>

#include <iostream>

#include <vector>

#include <list>

#include <queue>

#include <algorithm>

#include <string>

#include<fstream>

using namespace std;

#### Edge Class

class edge {

public:

int dest\_vertex\_id;

int weight;

edge() {

dest\_vertex\_id = -1;

weight = 0;

}

edge(int dv, int wt) {

dest\_vertex\_id = dv;

weight = wt;

}

void set\_dest\_vertex\_id(int dvi) { dest\_vertex\_id = dvi; }

void set\_weight(int w) { weight = w; }

int get\_dest\_vertex\_id() { return dest\_vertex\_id; }

int get\_weight() { return weight; }

};

#### Vertex Class

class vertex {

public:

int vertex\_id;

string place\_name;

list<edge> edgelist;

vertex() {

vertex\_id = -1;

place\_name = "";

}

void set\_vertex\_id(int v) { vertex\_id = v; }

void set\_place\_name(const string& name) { place\_name = name; }

string get\_place\_name() const { return place\_name; }

int get\_vertex\_id() { return vertex\_id; }

int get\_vertex\_id() const {

return vertex\_id;

}

list<edge>& get\_edge\_list() { return edgelist; }

void print\_edge\_list() {

cout << "[" << place\_name << "]: ";

for (auto it = edgelist.begin(); it != edgelist.end(); it++) {

cout << it->get\_dest\_vertex\_id() << "(" << it->get\_weight() << ") --> ";

}

cout << endl;

}

};

#### Graph Class

class graph {

public:

vector<vertex> mygraph;

void add\_vertex(const vertex& v) {

vertex new\_vertex = v; // Create a new vertex and copy the data from the passed vertex

new\_vertex.set\_vertex\_id(mygraph.size()); // Set the vertex ID based on the current size of mygraph

mygraph.push\_back(new\_vertex); // Add the new vertex to the graph

}

void add\_edge(const string& from, const string& to, int w) {

int from\_id = -1, to\_id = -1;

for (size\_t i = 0; i < mygraph.size(); ++i) {

if (mygraph[i].get\_place\_name() == from) {

from\_id = i;

}

if (mygraph[i].get\_place\_name() == to) {

to\_id = i;

}

}

if (from\_id != -1 && to\_id != -1) {

edge e1(to\_id, w);

mygraph[from\_id].edgelist.push\_back(e1);

edge e2(from\_id, w);

mygraph[to\_id].edgelist.push\_back(e2);

}

else {

cout << "Invalid place name(s) for edge: " << from << " - " << to << endl;

}

}

string findNearestHospital(const string& location) {

int location\_id = -1;

// Finding the location index in the graph

for (size\_t i = 0; i < mygraph.size(); ++i) {

if (mygraph[i].get\_place\_name() == location) {

location\_id = i;

break;

}

}

if (location\_id == -1) {

return "Location not found in the graph.";

}

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

vector<int> distTo(mygraph.size(), INT\_MAX);

distTo[location\_id] = 0;

pq.push({ 0, location\_id });

while (!pq.empty()) {

int node = pq.top().second;

int dis = pq.top().first;

pq.pop();

for (auto it : mygraph[node].get\_edge\_list()) {

int v = it.dest\_vertex\_id;

int w = it.weight;

if (dis + w < distTo[v]) {

distTo[v] = dis + w;

pq.push({ dis + w, v });

}

}

}

int minDist = INT\_MAX;

string nearestHospital;

// Finding the nearest hospital from the calculated distances

for (size\_t i = 0; i < mygraph.size(); ++i) {

if (mygraph[i].get\_place\_name().find("Hospital") != string::npos && distTo[i] < minDist) {

minDist = distTo[i];

nearestHospital = mygraph[i].get\_place\_name();

}

}

if (nearestHospital.empty()) {

return "No hospitals found nearby.";

}

return nearestHospital;

}

// ... (Other methods for graph operations)

};

class sol {

public:

int checkmain;

sol() : checkmain(0) {}

pair<vector<int>, vector<int>> dijkstra(int V, graph& g, int S, int dv) {

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

// Use a value that is less likely to cause overflow

vector<int> distTo(V, INT\_MAX);

vector<int> parent(V);

for (int i = 0; i < V; i++) {

parent[i] = i;

}

parent[S] = S;

distTo[S] = 0;

pq.push({ 0, S });

while (!pq.empty()) {

int node = pq.top().second;

int dis = pq.top().first;

pq.pop();

for (auto it : g.mygraph[node].get\_edge\_list()) {

int v = it.dest\_vertex\_id;

int w = it.weight;

if (dis + w < distTo[v]) {

distTo[v] = dis + w;

pq.push({ dis + w, v });

parent[v] = node;

}

}

}

// Debug print statements

// cout << "Distances: ";

//for (int i : distTo) {

// cout << i << " ";

// }

//cout << endl;

cout << "Parents: ";

for (int i : parent) {

cout << i << " ";

}

cout << endl;

vector<int> path;

int mynode = dv;

while (parent[mynode] != mynode) {

path.push\_back(mynode);

mynode = parent[mynode];

}

path.push\_back(S);

reverse(path.begin(), path.end());

// Debug print statement

cout << "Path: ";

for (int i : path) {

cout << i << " ";

}

cout << endl;

return { path, parent };

}

};

#### Feedback Class

class Feedback {

public:

int rating;

string comments;

Feedback() {

rating = 0;

comments = "";

}

void setRating(int r)

{

rating = r;

}

void setComments(const string& c)

{

comments = c;

}

int getRating() const

{

return rating;

}

const string& getComments() const

{

return comments;

}

};

#### Textbox Class

class TextBox {

public:

sf::Text text;

sf::Text label;

sf::RectangleShape box;

TextBox(sf::Font& font, unsigned int charSize, sf::Color textColor, sf::Color boxColor, float x, float y, float width, float height, const std::string& labelText) {

text.setFont(font);

text.setCharacterSize(charSize);

text.setFillColor(textColor);

label.setFont(font);

label.setCharacterSize(charSize);

label.setFillColor(sf::Color::White);

label.setString(labelText);

box.setSize(sf::Vector2f(width, height));

box.setFillColor(boxColor);

box.setOutlineThickness(2.0f);

box.setOutlineColor(sf::Color::White);

setPosition(x, y);

}

void setPosition(float x, float y) {

label.setPosition(x + 5, y - 30);

text.setPosition(x + 5, y + 5);

box.setPosition(x, y);

}

void handleEvent(const sf::Event& event) {

if (event.type == sf::Event::TextEntered) {

// Handle backspace

if (event.text.unicode == 8 && !text.getString().isEmpty()) {

std::string currentStr = text.getString().toAnsiString();

currentStr.pop\_back();

text.setString(currentStr);

}

// Handle regular character input

else if (event.text.unicode != 8 && event.text.unicode != '\r' && event.text.unicode != '\n') {

text.setString(text.getString() + static\_cast<char>(event.text.unicode));

}

}

else if (event.type == sf::Event::KeyPressed) {

if (event.key.code == sf::Keyboard::Enter) {

// Handle Enter key press here

// You may want to perform some action or switch to the next text box

}

}

}

void draw(sf::RenderWindow& window) {

window.draw(box);

window.draw(text);

window.draw(label);

}

void clear() {

text.setString("");

}

std::string getPlace() const {

return text.getString().toAnsiString();

}

};

void saveFeedback(const Feedback& feedback) {

ofstream file("feedback.txt", ios::app); // Open file in append mode

if (file.is\_open()) {

file << "Rating: " << feedback.getRating() << "\n";

file << "Comments: " << feedback.getComments() << "\n";

file << "------------------------\n"; // Separation for multiple feedback entries

file.close();

cout << "Feedback saved successfully!\n";

}

else {

cout << "Error: Unable to open the feedback file.\n";

}

}

void displayFeedback() {

ifstream file("feedback.txt");

if (file.is\_open()) {

string line;

while (getline(file, line)) {

cout << line << endl;

}

file.close();

}

else {

cout << "Error: Unable to open the feedback file.\n";

}

}

bool isMouseOverButton(const sf::RectangleShape& button, const sf::RenderWindow& window) {

sf::Vector2i mousePos = sf::Mouse::getPosition(window);

sf::FloatRect buttonBounds = button.getGlobalBounds();

return buttonBounds.contains(static\_cast<float>(mousePos.x), static\_cast<float>(mousePos.y));

}

void displayTextBoxes(sf::RenderWindow& window, graph& g1) {

// Load background image

sol obj;

vertex V;

sf::Texture bgTexture;

if (!bgTexture.loadFromFile("Map.jpg")) {

std::cerr << "Failed to load background image." << std::endl;

return;

}

sf::Sprite bgSprite(bgTexture);

bgSprite.setScale(static\_cast<float>(window.getSize().x) / bgTexture.getSize().x,

static\_cast<float>(window.getSize().y) / bgTexture.getSize().y);

// Create a font for text rendering

sf::Font font;

if (!font.loadFromFile("Roboto-Regular.ttf")) {

std::cerr << "Failed to load font." << std::endl;

return;

}

// Create text boxes for source and destination

TextBox sourceBox(font, 20, sf::Color::White, sf::Color(50, 50, 50), 20, 120, 300, 40, "Enter Source Location:");

TextBox destinationBox(font, 20, sf::Color::White, sf::Color(50, 50, 50), 20, 220, 300, 40, "Enter Destination Location:");

// Text for result display

sf::Text resultText("", font, 20);

resultText.setFillColor(sf::Color::White);

resultText.setPosition(20, 320);

// Flag to switch focus between text boxes

bool focusOnSource = true;

// Main loop

while (window.isOpen()) {

sf::Event event;

while (window.pollEvent(event)) {

if (event.type == sf::Event::Closed) {

window.close();

}

// Handle events for the focused text box

if (focusOnSource) {

sourceBox.handleEvent(event);

}

else {

destinationBox.handleEvent(event);

}

// If Enter is pressed, switch focus and clear the focused text box

if (event.type == sf::Event::KeyPressed && event.key.code == sf::Keyboard::Enter) {

if (focusOnSource) {

focusOnSource = false;

}

else {

std::string start = sourceBox.text.getString().toAnsiString();

std::string end = destinationBox.text.getString().toAnsiString();

// Find indices of start and end places

int start\_index = -1, end\_index = -1;

for (size\_t i = 0; i < g1.mygraph.size(); ++i) {

if (g1.mygraph[i].get\_place\_name() == start) {

start\_index = i;

}

if (g1.mygraph[i].get\_place\_name() == end) {

end\_index = i;

}

// Break out of the loop if both indices are found

if (start\_index != -1 && end\_index != -1) {

break;

}

}

if (start\_index != -1 && end\_index != -1) {

// Calculate shortest path using Dijkstra's algorithm

pair<vector<int>, vector<int>> result = obj.dijkstra(g1.mygraph.size(), g1, start\_index, end\_index);

vector<int> shortestPath = result.first;

// Display the shortest path on the SFML window

std::string pathText = "Shortest Path from " + start + " to " + end + ":\n";

for (int node : shortestPath) {

pathText += " -> " + g1.mygraph[node].get\_place\_name();

}

resultText.setString(pathText);

// Clear the text boxes for the next round

sourceBox.clear();

destinationBox.clear();

focusOnSource = true;

}

else {

resultText.setString("Invalid place name(s) entered.");

}

}

}

}

// Clear the window

window.clear();

// Draw background

window.draw(bgSprite);

// Draw text boxes and result text

sourceBox.draw(window);

destinationBox.draw(window);

window.draw(resultText);

// Display the window

window.display();

}

}

void displayPlaces(sf::RenderWindow& window, graph& g1) {

sf::Texture backgroundTexture;

if (!backgroundTexture.loadFromFile("MAP.jpg")) {

std::cout << "Error loading background image." << std::endl;

return;

}

bool isButtonHovered = false;

sf::Sprite background(backgroundTexture);

background.setScale(static\_cast<float>(window.getSize().x) / backgroundTexture.getSize().x,

static\_cast<float>(window.getSize().y) / backgroundTexture.getSize().y);

sf::Font font;

if (!font.loadFromFile("Roboto-Regular.ttf")) {

std::cout << "Error loading font." << std::endl;

return;

}

// Top box settings

const float topBoxHeight = 50.f;

const float topBoxYPos = 10.f;

sf::RectangleShape topBox;

topBox.setSize(sf::Vector2f(window.getSize().x, topBoxHeight));

topBox.setFillColor(sf::Color(100, 100, 200));

topBox.setPosition(0, topBoxYPos);

sf::Text topBoxText;

topBoxText.setFont(font);

topBoxText.setCharacterSize(20);

topBoxText.setFillColor(sf::Color::White);

topBoxText.setStyle(sf::Text::Bold);

topBoxText.setString("Available Places");

sf::FloatRect textBounds = topBoxText.getLocalBounds();

topBoxText.setOrigin(textBounds.left + textBounds.width / 2.f, textBounds.top + textBounds.height / 2.f);

topBoxText.setPosition(window.getSize().x / 2.f, topBoxYPos + topBoxHeight / 2.f);

std::vector<sf::Text> placeTexts;

std::vector<sf::RectangleShape> bubbleRectangles;

const float horizontalSpacing = 20.f;

const float verticalSpacing = 20.f;

float maxTextWidth = 0.f;

const float buttonWidth = 200.f;

const float buttonHeight = 50.f;

const float buttonXPos = (window.getSize().x - buttonWidth) / 2.f;

const float buttonYPos = window.getSize().y - 70.f;

sf::RectangleShape findPathButton;

findPathButton.setSize(sf::Vector2f(buttonWidth, buttonHeight));

findPathButton.setFillColor(sf::Color(50, 200, 50));

findPathButton.setOutlineColor(sf::Color(20, 100, 20));

findPathButton.setOutlineThickness(2.f);

findPathButton.setPosition(buttonXPos, buttonYPos);

sf::Text buttonLabel;

buttonLabel.setFont(font);

buttonLabel.setCharacterSize(20);

buttonLabel.setFillColor(sf::Color::White);

buttonLabel.setStyle(sf::Text::Bold);

buttonLabel.setString("Find Path");

sf::FloatRect labelBounds = buttonLabel.getLocalBounds();

buttonLabel.setOrigin(labelBounds.left + labelBounds.width / 2.f, labelBounds.top + labelBounds.height / 2.f);

buttonLabel.setPosition(buttonXPos + buttonWidth / 2.f, buttonYPos + buttonHeight / 2.f);

// Find the maximum text width

for (const vertex& v : g1.mygraph) {

sf::Text placeText;

placeText.setFont(font);

placeText.setCharacterSize(16);

placeText.setFillColor(sf::Color(0, 0, 0));

placeText.setStyle(sf::Text::Bold);

placeText.setString(v.get\_place\_name());

sf::FloatRect textBounds = placeText.getGlobalBounds();

if (textBounds.width > maxTextWidth) {

maxTextWidth = textBounds.width;

}

}

const float bubbleWidth = maxTextWidth + 20.f;

const float bubbleHeight = 60.f;

for (std::size\_t i = 0; i < g1.mygraph.size(); ++i) {

sf::Text placeText;

placeText.setFont(font);

placeText.setCharacterSize(16);

placeText.setFillColor(sf::Color(0, 0, 0));

placeText.setStyle(sf::Text::Bold);

placeText.setString(g1.mygraph[i].get\_place\_name());

sf::FloatRect textBounds = placeText.getGlobalBounds();

// Calculate the row and column indices

std::size\_t row = i / 3;

std::size\_t col = i % 3;

// Set the position of the bubble rectangle based on the row and column

float xPos = col \* (bubbleWidth + horizontalSpacing) + horizontalSpacing;

float yPos = row \* (bubbleHeight + verticalSpacing) + verticalSpacing + topBoxHeight + topBoxYPos;

sf::RectangleShape bubbleRect;

bubbleRect.setSize(sf::Vector2f(bubbleWidth, bubbleHeight));

bubbleRect.setFillColor(sf::Color(200, 200, 255));

bubbleRect.setOutlineColor(sf::Color(50, 50, 150));

bubbleRect.setOutlineThickness(2.f);

bubbleRect.setPosition(xPos, yPos);

bubbleRectangles.push\_back(bubbleRect);

// Center the text within the bubble rectangle

float textXPos = xPos + (bubbleWidth - textBounds.width) / 2.f;

float textYPos = yPos + (bubbleHeight - textBounds.height) / 2.f;

placeText.setPosition(textXPos, textYPos);

placeTexts.push\_back(placeText);

}

while (window.isOpen()) {

sf::Event event;

while (window.pollEvent(event)) {

if (event.type == sf::Event::Closed) {

window.close();

}

else if (event.type == sf::Event::MouseButtonReleased) {

sf::Vector2f mousePos = window.mapPixelToCoords(sf::Mouse::getPosition(window));

// Check if the Find Path button is clicked

if (findPathButton.getGlobalBounds().contains(mousePos)) {

// Execute the displayTextBoxes function

displayTextBoxes(window, g1);

}

// Check if any place bubble is clicked and perform relevant actions

for (std::size\_t i = 0; i < bubbleRectangles.size(); ++i) {

if (bubbleRectangles[i].getGlobalBounds().contains(mousePos)) {

// Perform actions for the clicked place

std::cout << "Clicked on place: " << g1.mygraph[i].get\_place\_name() << std::endl;

}

}

}

// Handle other events such as mouse clicks, button presses, etc.

// ...

}

window.clear();

// Draw background

window.draw(background);

// Draw top box

window.draw(topBox);

window.draw(topBoxText);

// Draw bubble rectangles

for (const sf::RectangleShape& bubbleRect : bubbleRectangles) {

window.draw(bubbleRect);

}

// Draw list of places

for (const sf::Text& placeText : placeTexts) {

window.draw(placeText);

}

isButtonHovered = isMouseOverButton(findPathButton, window);

findPathButton.setFillColor(isButtonHovered ? sf::Color(70, 220, 70) : sf::Color(50, 200, 50));

window.draw(findPathButton);

window.draw(buttonLabel);

window.display();

}

}