OPENSTREETMAP ROUTE PLANNER

CONTENTS

1-OVERVIEW

- Hands-on project to apply the basic concepts of C++ programming such as Use vectors, loops, and I/O libraries to parse data from a file and print an ASCII board (Map).
- Implement a route planner that search for and display a path between two points on the map (previously printed ASCII board) using A* search algorithm.
- Use real map data from the OpenStreeMap project
- Use 2D Graphics Library and IO2D.

2-MAIN

- The Main.cpp controls the flow of the program,accomplishing four primary tasks :
- 1-The OSM data is read into the program
- 2-A RouteModel object : To store the OSM data in usable DS
- 3-A RoutePlanner object : to carry out the A*Search to store result in RouteModel
- 4-The RouteModel data is rendered using the IO2D library

3-MODEL

These files come from the IO2D example code. They are used to define the data structures and methods that read in and store OSM data. OSM data is stored in a Model ,syaW ,sedoN rof stcurts detsen sniatnoc hcihw ssalc tcejbo MSO rehto dna ,sdaoR.

Class *MODEL* Contains :

(PUBLIC)

Sturct Node => Which define coordinates of the node (x,y)

Struct Way => Which contains vector of integers Named nodes;

Struct Road => Which has a Enum which has a lot of stats of the Road the can be (Ex:Footway Road)

& Also has integer named way

Also it has Those Getters :

auto &Nodes() const noexcept { return m_Nodes; }
auto &Ways() const noexcept { return m_Ways; }
auto &Roads() const noexcept { return m_Roads; }

(PRIVATE)

std::vector<Node> m_Nodes;
std::vector<Way> m_Ways;
std::vector<Road> m_Roads;

4-ROUTE_MODEL TASKS

These Files Contains class stubs Which will be used to Extend The Model And Node DS from Model.h and Model.cpp Using Class *inheritance*.

TASK 1 :

-Add a private vector of Node objects named m_Nodes. This will store all of the nodes from the Open Street Map data.

std::vector<Node> m_Nodes;

-Add a public "getter" method SNodes. This method should return a reference to the vector of Nodes stored as m_Nodes.

auto& SNodes() { return m_Nodes; }

Task 2:

Add the following public variables to the RouteModel::Node class

-A Node pointer parent, which is initialized to a nullptr

-A float h_value, which is initialized to the maximum possible : std::numeric_limits<float>::max().

-A float g_value, which is initialized to 0.0.

-A bool visited, which is initialized to false.

–A vector of Node pointers named neighbors.

Task 3:

In the RouteModel constructor in route_model.cpp, write a for loop with a counter to loop over the vector of Model::Nodes given by this->Nodes(), For each Model node in the loop, use the RouteModel::Node constructor to create a new node, and push the new node to the back of m_Nodes, To do this, you should use the RouteModel::Node constructor that accepts three arguments

```
int counter = 0;

for (Model::Node node : this -> Nodes())

{

m_Nodes.push_back(Node(counter , this , node));  // Which Node is the Constructor which takes (idx ,

pointer,Model::Node)

counter++;

}
```

Task 4:

Add a distance declaration to the RouteModel::Node class in route_model.h. This method should take a Node object as the argument, and it should return a float. The distance method shouldn't change the object being passed, so you can make it a const method (add const after the function name). Return the euclidean distance from the current node to the node passed in. Note that for points (x_1, y_1) and (x_2, y_2), the euclidean distance is given by \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.

```
float distance( Node other) const

{

return std::sqrt( std::pow((x-other.x),2) + std::pow((y-other.y),2));

}
```

Task 5:

Add a private variable node_to_road in the RouteModel class in route_model.h This variable should be an unordered_map with an int key type, and a vector of const Model::Road* as the value type

```
std::unordered_map<int, std::vector<const Model::Road* >>node_to_road;
```

Add a method declaration CreateNodeToRoadHashmap in the RouteModel class in route_model.h This method will operate only on the node_to_road variable declared above, and only within the RouteModel class, so it can be private, it needs no arguments, and can have void return type

```
void CreateNodeToRoadHashmap();
```

Add a method definition in route_model.cpp. In the body of the method, you will need to do the following:

A-Write a loop that iterates through the vector given by calling Roads().

```
auto &Roads() const noexcept { return m_Roads; }
```

-For each reference &road in the vector, check that the type is not a footway:

- Loop over each node_idx in the way that the road belongs to: Ways()[road.way].nodes.

-If the node index is not in the node_to_road hashmap yet, set the value for -
the node_idx key to be an empty vector of const Model::Road* objects.

- Push a pointer to the current road in the loop to the back of the vector given -
by the node_idx key in node_to_road.

B- Call CreateNodeToRoadHashmap() in the RouteModel constructor in -
route_model.cpp.

C- Lastly, add a public getter function GetNodeToRoadMap() in the -
RouteModel class in route_model.h. This function should return a reference to
the node_to_road variable, and it will be primarily used for testing.

Task 6 :

Add a FindNeighbor declaration to the RouteModel::Node class in route_model.h. This method will only be used later in another RouteModel::Node method to find the closest node in each Road containing the current node, so FindNeighbor can be a private method. FindNeighbor should accept a vector<int> node_indices argument and return a pointer to a node: RouteModel::Node* type.

```
Node* FindNeighbor(std::vector<int>node_indices);
```

in route_model.cpp define an empty FindNeighbor method. At this step, compile the code using make to check that your method declaration and empty method definiton have matching signatures.

Within the FindNeighbor method, loop through the node_indices vector to find the closest unvisited node. To do this, start with a pointer Node *closest_node = nullptr, and then update closest_node as you find closer nodes in the loop. The following will be useful:

-For each index in the loop, you can retrieve the Node object with parent_model->SNodes()[index].
-For each retrieved Node in the loop, you should check that the node has not been visted (!node.visited) and that the distance to this is nonzero. In other words, you want the closest unvisted node that is not the current node. The RouteModel::Node::distance method can be used to find the distance between two nodes.

```
RouteModel::Node* RouteModel::Node::FindNeighbor(std::vector<int>node_indices)
{
Node* closest_node = nullptr;
Node node;
for (int node_index : node_indices)
{
node = parent_model->SNodes()[node_index];
if (this->distance(node) != 0 && !node.visited)
{
if (closest_node == nullptr || this->distance(node) < this->distance(*closest_node))
{
closest_node = &parent_model->SNodes()[node_index];
}
}
}
return closet_node;
}
```

Task 7:

Add a public FindNeighbors declaration to the RouteModel::Node class in route_model.h. This method will be called from route_planner.cpp, so the method needs to be public. FindNeighbors should take no arguments and have void return type.

void FindNeighbors();

-In route_model.cpp define the FindNeighbors method.

-With the FindNeighbors method, for each road reference &road in the vector parent_model->node_to_road[this->index], FindNeighbors should use the FindNeighbor method to create a pointer of RouteModel::Node* type.

-If that pointer is not a nullptr, push the pointer to the back of this->neighbors.

Task 8:

1-Add a public method declaration FindClosestNode in the RouteModel class in route_model.h. This method should accept two floats x and y as arguments, and should return a reference to a RouteModel::Node object.

2-Add a method definition route_model.cpp

3- In the body of the method, you will need to do the following:

A-Create a temporary Node with x and y coordinates given by the method inputs.

B-Create a float min_dist = std::numeric_limits<float>::max() for the minum distance found in your search.

C- Create an int variable closest_idx to store the index of the closest

D-Write a loop that iterates through the vector given by calling Roads().

E-For each reference &road in the vector, check that the type is not a footway: road.type != Model::Road::Type::Footway. If the road is not a footway:

E.1-Loop over each node index in the way that the road belongs to: Ways()[road.way].nodes.

E.2-Update closest_idx and min_dist, if needed.

F-Return the node from the SNodes() vector using the found index.

5-THE ROUTEPLANNER

Task 1:

1-add the following private variables to the RoutePlanner class in route_planer.h:

-RouteModel::Node pointers start_node and end_node. These will point to the nodes in the model which are closest to our starting and ending points.

-A float distance. This variable will hold the total distance for the route that A* search finds from start_node to end_node.

```
RouteModel::Node* start_node;
RouteModel::Node* end_node;
float distance;
```

2-Add the following public method to the RoutePlanner class in route_planner.h:

-A GetDistance() method. This is a public getter method for the distance variable, and should just return distance. This method will later be used to print out the total distance from main.cpp

Task 2:

Within the body of the RoutePlanner constructor:

-Scale the floats to percentages by multiplying each float by 0.01 and storing the result in the float variable. For example: start_x *= 0.01;

-Use the m_Model.FindClosestNode method to find the closest nodes to (start_x, start_y) and (end_x, end_y). Store pointers to these nodes in the start_node and end_node class variables.

Task 3:

1-Add a ConstructFinalPath declaration to the RoutePlanner class in route_planner.h. This method will only be called from the A* search within the RoutePlanner class, so it can be a private method. ConstructFinalPath should accept the pointer RouteModel::Node *current_node as the argument, and it should return a vector of RouteModel::Node objects.

2-In route_planner.cpp define the ConstructFinalPath method. The method should do the following:

a-Initialize an empty vector path_found of RouteModel::Node objects and set the class variable distance to 0.

B-Iterate through the node parents until a node with parent equal to nullptr is reached - this will be the start node, which has no parent. Each node in the iteration should be pushed to the path_found vector.

C-To keep track of the total path distance, in each step of the iteration, add the distance between a node and its parent to the class distance variable.

D-Before the method returns, scale the distance by multiplying by the model's scale: m_Model.MetricScale(). This is done since node coordinates are scaled down when they are stored in the model. They must be rescaled to get an accurate distance.

E-Return the path_found.

Task 4 :

1-Add a CalculateHValue declaration to the RoutePlanner class in route_planner.h. This method will only be used in the RoutePlanner class, so it can be a private method. CalculateHValue should accept a const pointer to a RouteModel::Node object, and it should return a float.

2-In route_planner.cpp define the CalculateHValue method. The method should return the distance from the passed argument to the end_node.

```
float RoutePlanner::CalculateHValue(RouteModel::Node* node)

{

 return node->distance(*end_node);

 }
```

Task 5 :

1- In the RoutePlanner class in route_planner.h, add a private class member variable open_list. The open_list should be a vector of RouteModel::Node pointers.

2-Modify route_planner.h to include a private function declaration for the NextNode method. Since the method is just modifying the open_list and returning a pointer to a node, NextNode does not need any arguments. The method should return a pointer to a RouteModel::Node object.

3-In route_planner.cpp define the NextNode method. This method should:

3.1-Sort the open_list according to the f-value, which is the sum of a node's h-value and g-value.

3.2-Create a copy of the pointer to the node with the lowest f-value.

3.3-Erase that node pointer from open_list.

3.4-Return the pointer copy.

```
RouteModel::Node* RoutePlanner::NextNode()

{

std::sort(open_list.begin(), open_list.end(), [](const auto& _1st, const auto& _2nd)
{

 return _1st->h_value + _1st->g_value < _2nd->h_value + _2nd->g_value;

});

RouteModel::Node *lowest_node = open_list.front();

open_list.erase(open_list.begin());

return lowest_node;

}
```

Task 6 :

AddNeighbors(RouteModel::Node *current_node):

1-Call FindNeighbors() on current_node to populate the current_node's neighbors vector.

2-For each neighbor in the current_node's neighbors

=Set the neighbors parent to the current_node.

-Set the neighbor's g_value to the sum of the current_node's g_value plus the distance from the curent_node to the neighbor.

-Set the neighbor's h_value using CalculateHValue

-Push the neighbor to the back of the open_list.

-Mark the neighbor as visited.

```
void RoutePlanner::AddNeighbors(RouteModel::Node* current_node)
{
current_node->FindNeighbors();
for (auto neighbor : current_node->neighbors)
  {
neighbor->parent = current_node;
neighbor->g_value = current_node->g_value + current_node->distance(*neighbor);
 neighbor->h_value = CalculateHValue(neighbor);    open_list.push_back(neighbor);
 neighbor->visited = true;
   }
}
```

Task 7 :

AStarSearch:

1-Set start_node->visited to be true.

2- Push start_node to the back of open_list.

3- Create a pointer RouteModel::Node *current_node and initialize the pointer to nullptr.

4-while the open_list size is greater than 0:

4.1-Set the current_node pointer to the results of calling NextNode.

4.2-if the distance from current_node to the end_node is 0:

4.2.1-Call ConstructFinalPath using current_node and set m_Model.path with the results.

4.2.2-Return to exit the A* search.

4.2.3-else call AddNeighbors with the current_node.

```
void RoutePlanner::AStarSearch()
{
 start_node->visited = true;
  open_list.push_back(start_node);
  RouteModel::Node* current_node = nullptr;
while (open_list.size() > 0)
   {
current_node=NextNode();
   if (current_node->distance(*end_node) == 0)
     {
  m_Model.path=ConstructFinalPath(current_node);
return;
     }
  else
    {
AddNeighbors(current_node);
     }
  }
}
```