CMSC 341 - Project 1: Network of Charging Stations - Fall 2021

Due: Tuesday, Oct 05, before 9:00 pm

Addenda

- 09/29/2021 - Clarification: the linked list in this project is used to store all nodes, to preserve the integrity of the linked list we use `m_next` member variable. All nodes should be in the linked list, and there is only one instance of every node in this project. The concept of graph is realized using the data in every node. Every node stores up to 4 pointers to other nodes, those pointers are stored in `m_north, m_east, m_south, and m_west`.
- 09/29/2021 - Clarification: in overloaded assignment operator there is no need to preserve the order of RHS nodes in the current object. However, copying all nodes to current object is important.
- 09/24/2021 - Clarification: the specifications for the `Graph::buildGraph(...)` function has been modified to clarify the functionality of this function.
- 09/15/2021 - Modification: The file `graph.h` has been modified to remove Wreorder warnings. You can download the latest from the links on this page.

Objectives

- Implementing a linked list container.
- Implementing a graph data structure.
- Implemeting a recursive graph traversal algorithm.

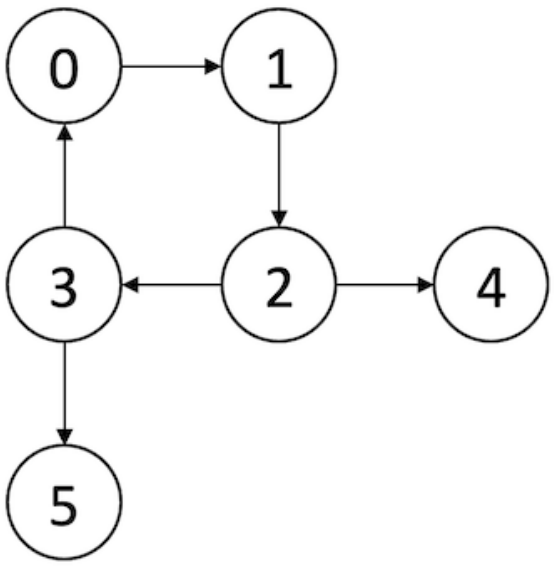
Introduction

According to the International Energy Agency in 2020 there were 1.8 million electric vehicles registered in the US. This is compared to the year 2016 with 300,000 registered electric vehicles. With the growth rate in sales, a startup company saw the opportunity to create a network of charging stations nationwide. You are assigned a task to develop an app that the drivers can use to plan their trips based on the available charging stations. The app finds a path between a start point and an end point.

Graph

To present the network of connected nodes we may use a graph. Every node has a unique name. In a graph the connections between nodes are called edges. An edge can have a direction to specify the direction of move from a node to a neighbor node. The neighbor nodes are called adjacent nodes.

The following figure presents a graph. In this graph one may move from node 2 to node 4, but there is no path from node 4 to node 2. Another observation is that one may start from node 0 and can reach to node 0 by passing through nodes 1, 2, and 3. That is a loop. Every node can connect to 4 neighbor nodes which are located at north, east, south, and west of the node.



In the project every node of a graph represents a charging station. The program should find all nodes that one needs to pass in order to reach a destination. For example, if a driver wants to go to node 5 from node 0, the program finds and reports the path 0 1 2 3 5. There may be more than one possible path between two nodes, in the current project any correct path is acceptable. In this project we are not trying to find the shortest or longest path. The following algorithm is a sample traversal method for finding the answer:

1. visit node 0 and tag it as visited,

2. from node 0 there is only 1 connection, then move to node 1, and tag node 1 as visited,
3. from node 1 there is only 1 connection, then move to node 2, and tag node 2 as visited,
4. from node 2 there are two connections, keep trying all connections until reaching the destination,
5. from node 2 move to node 4, tag node 4 as visited,
6. node 4 is a dead end, there is no connection, go back to node 2, this is called backtracking,
7. from node 2 move to node 3, tag node 3 as visited,
8. from node 3 there are multiple connections, try all until reaching the destination,
9. from node 3 move to node 0, it is already visited, it is a loop, there is no point to continue, backtrack to node 3,
10. from node 3 move to node 5, it is the destination, the problem is solved.

Input to Project

To encode this graph so that it is easy to read into a program, we will list each node along with its reachable neighbors. The following figure presents the data file for the above graph. The first line of the file shows the total number of nodes. The nodes data in the file are presented in 5 columns. The first column represents the node number, the second column represents the node number at north position, the third column represents the node number at east position, the fourth number represents the node number at south position, and the fifth number represents the node number at west position. For example, node 3 connects to node 0 at north. It is not connected to any node at its east side, this is indicated by -1. It is connected to node 5 at south, and it is not connected to any node at west.

```
6
0 -1  1 -1 -1
1 -1 -1  2 -1
2 -1  4 -1  3
3  0 -1  5 -1
4 -1 -1 -1 -1
5 -1 -1 -1 -1
```

The following list presents the assumptions about the data in the file:

- There is no edge from a node to itself.
- From any node in any direction there is only one edge out.

Assignment

Your assignment is to implement a graph data structure. The graph nodes will be stored in a linked list.

For this project, you are provided with the skeleton .h and .cpp files and a sample driver:

- graph.h - Interface for the Node and Graph classes.
- graph.cpp - This file will be completed and submitted.
- data.txt - The data file for the sample graph. You can use this data for development.
- testdata.txt - The data file for the sample graph. You use this data for testing.
- driver.cpp - A sample driver program.
- driver.txt - Sample output from driver.cpp.

Additionally, you are responsible for thoroughly testing your program. Your test program, `mytest.cpp`, must be submitted along with other files. For grading purposes, your implementation will be tested on input data of varying sizes, including very large data. Your submission will also be checked for memory leaks and memory errors.

Specifications

There are two classes in this project. The Node class stores the information for a node. The Graph class stores the information for a graph. A Graph object contains multiple Node objects.

Class Node

The implementation of this class is provided. You are not allowed to modify this class. The Node class implements a node that can be used in a linked list. Every node has a pointer to the next node in the linked list. This pointer is stored in the m_next member variable. Moreover, every object of this class can point to 4 other node objects which are represented by m_north, m_east, m_south, and m_west member variables. This structure allows for traversing across multiple nodes in different directions using the links.

Class Graph

This class implements a graph data structure. You need to implement the class. The graph nodes are stored in a linked list which is presented by the pointer to its first node. The pointer to the first node is stored in the m_head member variable. The member variable is a Node pointer. A Graph object reads a data file, creates the nodes, initializes the nodes and inserts them into the linked list.

Graph::Graph()	Default constructor creates an empty object and initializes member variables.
Graph::Graph(string dataFile)	Alternative constructor creates a graph using the information in dataFile. You can call the function loadData() here.
Graph::~Graph()	Destructor deallocate all memory and reinitialize member variables.
void Graph::loadData()	This function is implemented. you do not need to modify this function. It can be called in the alternative constructor to build the graph.
void Graph::insert(int node, int n, int e, int s, int w)	This function creates nodes and insert them into the linked list. The node parameter presents the value for the node to be inserted. Moreover, it uses the parameters n, e, s, and w to initialize the required fields of the node. Every node we read from data file is inserted only once in the linked list. Then, if a node is already in the list, we only update its member variables using the information. Here is a sample algorithm: 1. Read the line “0 -1 1 -1 -1”, the node with 0 value is not in the list. 2. Allocate memory for the node, and set its value. Since the node to its north is -1, then there is no need to initialize its m_north variable to a pointer, it should be only initialized to nullPtr. 3. The east pointer of node 0 points to the node 1. There is no node 1 in the list. Allocate memory and insert it into the list. 4. Initialize the m_east member variable of the node 0 with the pointer for node 1. And continue processing the information for node 0. 5. At next line of data file, read “1 -1 -1 2 -1”. Node 1 is already inserted into the list. Then, its information should be processed. The member variables m_north and m_east are initialized to nullPtr. 6. The south pointer of node 1 should point to node 2. If the node 2 is not in the list, create it and insert it. Then, initialize the m_south member variable of node 1 to the pointer for node 2. 7. Continue the insertion process through the end of information in the data file.
void Graph::insertAtHead(Node * aNode)	This function insertes aNode at the head of linked list and unpdates the required pointers to maintain the linked list. This can be called by the insert function whenever there is a need to insert a new node.
Node * Graph::findNode(int nodeValue)	This function traverse the linked list to find the node having nodeValue. If it finds the node it returns the pointer to the node, otherwise it returns nullPtr.
bool Graph::findPath(int start, int end)	This public function is the interface to the class. The user calls this function to find a path from the start node to the end node. The path will be stored in the member variable m_path. For finding every new path the information about previous path such as visited nodes or m_path should be cleared.
bool Graph::findPath(Node* aNode, int end)	This private function is a helper that can be used recursively to

	traverse the nodes. Originally it can be called on the pointer to the start node. Once in a node, it tags the node as visited, then it tries all possible connections, and so on. If it reaches a visited node, it backtracks. Backtrack means the function returns from that node with false return value. The reasons for backtracking are to avoid dead ends or loops. If the current node is newly visited, the function can record the nodes in the m_path variable and moves on. The sample algorithm provided above shows how the traversal can work.
void Graph::dump()	This function prints out the path to the output in a specific format. The format of the output is presented in the sample output.
void Graph::clearResult()	This function clears the path found.
void Graph::clearVisited()	This function resets the visited flag for all Node objects. It sets m_visited to false;
void Graph::buildGraph(string file)	This function builds a new graph using the information in file. This function reads a new input file and creates a new graph based on the new data. It first clears all data related to the current graph in the object, then it builds a new graph in the same object.
void Graph::clearGraph()	This function deallocates all memory and reinitializes all member variables. It makes the object an empty one.
const Graph & Graph::operator=(const Graph & rhs)	The assignment operator, creates a deep copy of rhs. **Reminder:** a deep copy means the current object has the same information as rhs object, however, it has its own memory allocated. Moreover, an assignment operator needs protection against self-assignment.

Requirement: The class declarations (`Node`) and (`Graph`) and provided function implementations in `graph.cpp` may not be modified in any way. No additional libraries may be used, but additional `using` statements are permitted. Private helper functions may be added, but must be declared in the private section of the `Graph` class. There is a comment indicating where private helper fuction declarations should be written.

Requirement: No STL containers or additional libraries may be used except the ones that are already in the files. STL containers or additional libraries may be used in your tests, i.e. mytest.cpp.

Requirement: Your code should not have any memory leaks or memory errors.

Requirement: Follow all coding standards as decribed on the C++ Coding Standards. In particular, indentations and meaningful comments are important.

Testing

Following is a non-exhaustive list of tests to perform on your implementation.

Note: Testing incrementally makes finding bugs easier. Once you finish a function and it is testable, make sure it is working correctly.

Testing Graph class:

- Test building a graph by the alternative constructor for a normal case.
- Test creating an empty graph object.
- Test building a graph in an empty Graph object.
- Test finding a path which does not exist in the graph.
- Test finding a path which exists in the graph.
- Test finding a path from a node to itself.
- Test finding a path in which the start node does not exist.
- Test finding a path in which the end node does not exist.
- Test Graph class assignment operator for a normal case and an empty graph object. Your test should check whether a deep copy is created. There are two ways of doing this. You can compare the list pointers, they should not match. Then, you compare all values in the corresponding nodes, all values should match. Another way of checking on deep copy is to clear the rhs object, and check whether the current object contains data. Either way is acceptable.

Memory leaks and errors:

- Run your test program in `valgrind`; check that there are no memory leaks or errors.
  Note: If `valgrind` finds memory errors, compile your code with the -g option to enable debugging support and then re-run `valgrind` with the `-s` and `--track-origins=yes` options. valgrind will show you the lines numbers where the errors are detected and can usually tell you which line is causing the error.
- Never ignore warnings. They are a major source of errors in a program.

What to Submit

You must submit the following files to the `proj1` submit directory:

- `graph.h`
- `graph.cpp`
- `mytest.cpp` (**Note:** This file contains the declaration and implementation of your Tester class as well as all your test cases.)
- `testdata.txt` (**Note:** You should not modify this file. Your test file should use the name of this file.)

If you followed the instructions in the Project Submission page to set up your directories, you can submit your code using the following command:

```
cp graph.h graph.cpp mytest.cpp testdata.txt ~/cs341proj/proj1/
```

Grading Rubric

The following presents a course rubric. It shows how a submitted project might lose points.

- Conforming to coding standards make about 10% of the grade.
- Correctness and completeness of your test cases (mytest.cpp) make about 15% of the grade.
- Passing tests make about 30% of the grade.

If the submitted project is in a state that receives the deduction for all above items, it will be graded for efforts. The grade will depend on the required efforts to complete such a work.
