Homework Assignment 6

Submit Assignment

Due Tuesday by 5pm **Points** 100 **Submitting** a website url

For this assignment, you will write a program to load a graph from a text file and then calculate a path that visits all of the vertices (i.e. nodes) once. This is commonly referred to as the traveling salesperson problem.

User Requirements

Package delivery, sales routes, and other similar scenarios are quite complex problems. In this case, the user is wanting to find the most efficient route to visit all of the package drop-off points in this region's cities. The user wants to only visit each city once, with the exception that the user wants to return to the origin city where they started from. This is commonly referred to as the traveling salesperson problem. We can model this problem by using a weighted, undirected graph. Each node or vertex is a city, and each edge weight represents the distance between two cities. We must visit each city exactly once and finish at the city where we started. This can be referred to as a *hamiltonian cycle* or a *tour*.

Overall, this is a very difficult problem to solve. It belongs to a special subset of problems in computer science referred to as **NP-hard**. We can approximate the solution by finding a **Minimum Spanning Tree** (MST). A MST is a subset of edges of a weighted undirected graph that connects all of the vertices without cycles and has the smallest possible total edge weight. By doing a preorder tree traversal on the MST, we can get a good approximation of the solution to the traveling salesperson problem.

File Format

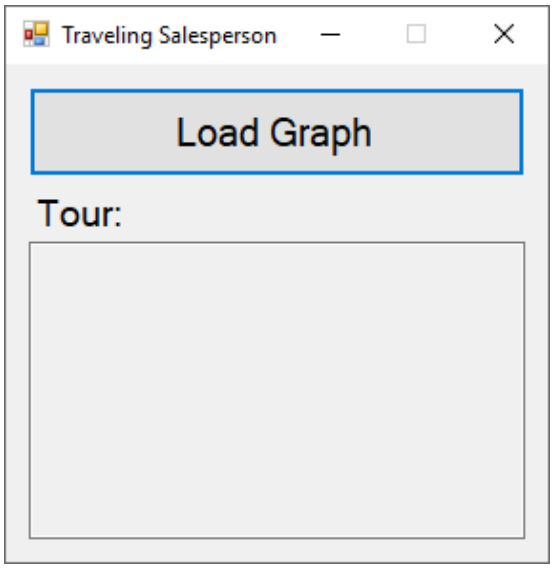
In the **Data** directory of the repository, there are a series of text files that contain complete graphs representing groups of cities and the distances between them. Note that a complete graph is a graph where each node is connected to all other nodes. The first line of the file is the **number** of cities (vertices) in the graph and then each line following represents an edge. Each edge has three parts in the following format: **source** (string datatype),**destination** (string datatype),**distance** (int datatype).

Starting the Assignment

You may use either Visual Studio Enterprise 2015 or Visual Studio Enterprise 2017 to do this assignment. Our experience so far this semester has suggested that VS 2015 might be more reliable, particularly with respect to source control (i.e., Git). If you do use VS 2017 on your own PC, we recommend installing all the updates (**note:** the lab machines don't necessarily have all the updates installed). The Community versions are not recommended.

Begin by creating a GitHub repository using **this link (https://classroom.github.com/a/TFA3K2Uo)** and clone it to your local machine. This repository contains a new Windows Forms Application and an executable called **hw6.exe** that demonstrates the operation of the program.

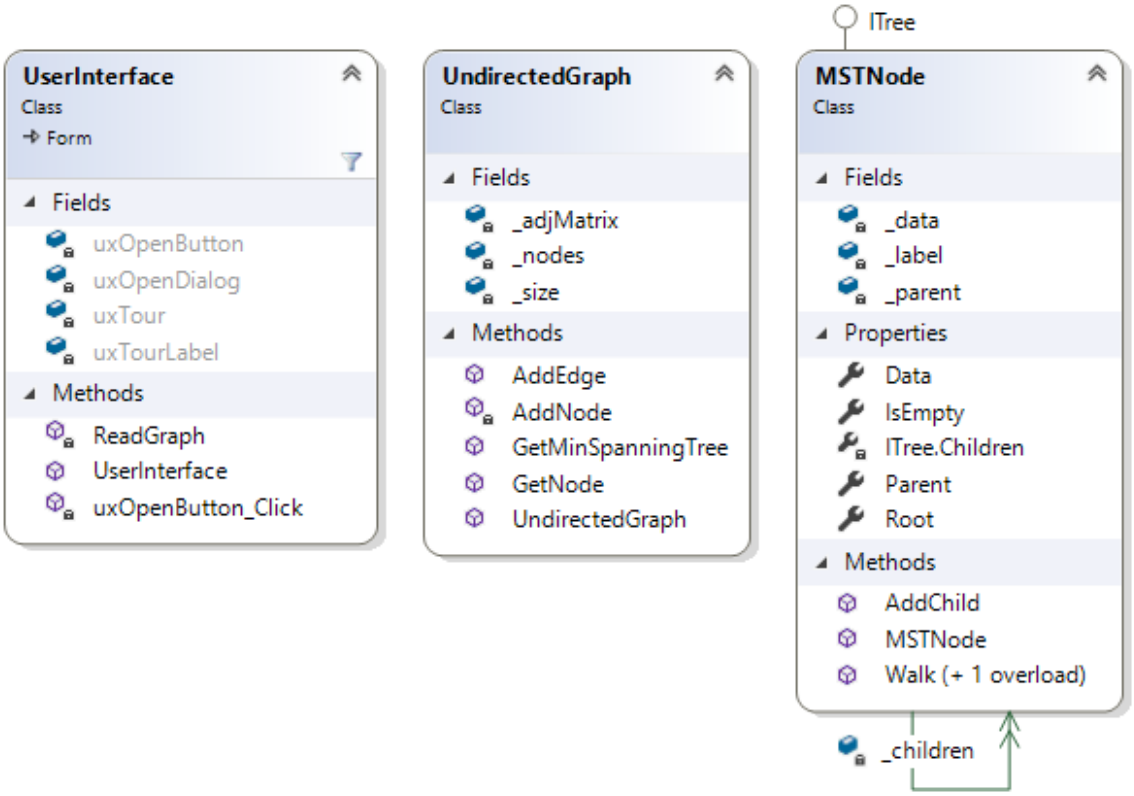
User Interface



The interface for this assignment is pretty straight forward. There is a button which should open an **OpenFileDialog** box to load a graph text file. Once the graph is fully processed, the MST created from the graph should be displayed using the **TreeViewer** as done in previous assignments and labs. It should also give the tour in the **TextBox** (see code requirements for how to format the tour). The **TextBox** should be labeled, and should also have the **AcceptsReturn, ReadOnly,** and **MultiLine** properties set to **true**. The **Label** and **Button** should have **Microsoft Sans Serif, 14.75pt** font. The form should not be able to be resized.

Software Architecture

See the following UML diagram below for the classes that you will be making.



Code Requirements

Your program should be organized into the following classes:

MSTNode

This class represents a node in a minimum spanning tree. The class itself implements the **ITree** interface to enable the nodes to be displayed using the **TreeViewer2** library included in the project. The class has the following fields and properties (note that the Children, IsEmpty, and Root properties are required by the ITree interface, and used for drawing the tree):

- **private int _data:**
  - the weight of the graph edge that took us to this node.
- **private int _parent:**
  - the array index from the adjacency matrix of the parent node
- **private string _label:**
  - the string representation of the node
- **private List<MSTNode> _children:**
  - the children of this node
- **public int Data:**
  - public property to get and set the private **_data** field
- **public int Parent:**
  - public property to get and set the private **_parent** field
- **public object Root:**

    - public property for the ITree interface to retrieve the representation of the node. This should return a **string** that has the label of the node combined with the data in parentheses, for example: **A (4)**
- **ITree[] ITree.Children:**
    - property for the ITree interface to return the children of this node. To do so, simply convert the **_children** field to an array and return.
- **public bool IsEmpty:**
    - public property that returns whether or not the node is empty. For this project, it should just return false since we never have any empty nodes.

This class also has the following methods:

- **public MSTNode(int data, int parent, string label):**
    - This is the public constructor for the class. It should simply set the corresponding private fields using the parameters.
- **public void AddChild(MSTNode child):**
    - This method should add the given child to the **_children** list.
- **private void Walk(StringBuilder sb):**
    - This is a *recursive* helper function to do a *pre-order depth-first* walk on the children of the MST. You can think of a walk as a kind of traversal. This should go in *pre-order* so the root must be processed first, followed by the left-most child. Once the left-most child is processed, then it should continue to the next child in order. It should continue the *pre-order* walk that is described in the main **Walk** function and append the results to the given **StringBuilder**.
- **public string Walk():**
    - This method initiates the *pre-order depth-first* walk of the MST tree described in the previous **Walk** method. This method should only be called on the root node of the tree. This method should start a **StringBuilder** to accumulate the walk as each node is processed. To recursively process a child, call the overloaded **Walk** method that was described previously. Once you are done processing each child of the root node, add the root node to the walk (using the string builder) to make the salesperson go back to the origin city (the root node). The method should return a string representing the completed walk. This string should be representative of a *hamiltonian cycle* that was described earlier. The format should resemble the text in the **uxTour** TextBox shown in the Testing section. Note that to add a new line in your string, use **Environment.NewLine.**

UndirectedGraph

This class implements a weighted, undirected graph by using an adjacency matrix. Each node will be an integer index (0, 1, 2, etc.) and will have a string name/representation. The class has the following fields:

- **private int[,] _adjMatrix:**
    - a two dimensional array to store the adjacency matrix representing the graph. The row indices are the source nodes and the column indices are the edges. The values stored are the edge weights. This is different than the traditional adjacency matrix described in the **readings**

(http://people.cs.ksu.edu/~rhowell/DataStructures/graphs/impl.html) . Node pairs that do not have an edge have a weight of 0. This means that the graph cannot have any edge weights of 0.

- **private List<string> _nodes:**
  - the string representation of each node. The index in the list corresponds to a node in the adjacency matrix.
- **private int _size:**
  - the number of nodes in the graph

This class also has the following methods:

- **public UndirectedGraph(int size):**
  - This method sets the **_size** field and initializes the adjacency matrix to a new two-dimensional array with **_size** as the number of rows and columns.
- **private void AddNode(string data):**
  - This function adds a corresponding string representation of a node to the node list **_nodes**. It should throw a **InvalidOperationException** to indicate that the graph is full if the number of nodes is the same as the size of the graph before adding.
- **public string GetNode(int index):**
  - This function simply returns the string representation of a node at the given index.
- **public void AddEdge(string source, string destination, int weight):**
  - Note that since this is an undirected graph, the edge weight should be stored at both the source/destination and the destination/source.
  - Note that you should try to find/add the source index first in order to be able to match the test output. Otherwise, node labels may be switched around.
  - This method adds the edge given by the source, destination, and weight parameters to the graph. If either node has never been seen before, it should be added to the list of **_nodes** by calling the **AddNode** method. If a node has already been seen, it should be in the **_nodes** list. After the indices of the source and destination nodes are found, the weight should be stored at that location in the adjacency matrix.
- **public MSTNode GetMinSpanningTree():**
  - This method calculates the minimum cost spanning tree of the graph by using **Prim's algorithm**. Prim's algorithm builds an MST by adding one node at a time. The node added is determined by the weights of the edges connecting nodes in tree to nodes not yet in the tree. Of these edges, the one with minimum weight will be selected - its destination node will be made a child of its source node. This MST will be used in the **Walk** methods described in the **MSTNode** class to generate a hamiltonian cycle or tour to solve the traveling salesperson problem. The algorithm for Prim's is as follows:
    - Initialize two arrays of length **_size**. One should be an array of **MSTNode**s that will hold the minimum spanning tree (referred to as **tree** in this algorithm description). The other should be a boolean array that keeps track of which nodes have been added to the **tree** (this array will be referred to as **added** in this algorithm description).
    - Initialize each element (except the 0th element, this will be the root of the **tree**) of the MSTNode[ ] to a new **MSTNode** with Int32.MaxValue as the data, 0 as the parent, and the corresponding node

string representation as the label. Remember that the **Data** property is the weight of the edge that took us to this tree node. Therefore we initialize all Data values to the max possible value to make sure we process only the tree node that has the smallest **Data** value on each iteration of the loop below. The Data value will be updated as the graph is traversed.

- Initialize the root of **tree** to a new **MSTNode** with 0 as the data, -1 as the parent (the root can't have any parents), and the corresponding node string representation as the label.
- Repeat for the size of the graph:
  - Find the index of the unvisited node having the smallest Data value. This index is referred to as **minIndex** in this description.
  - Set the **minIndex** to being **added**.
  - If we are not on the first iteration of the loop, add the node contained at the **minIndex** from the tree as a child to its parent. Note that this helps maintain the correct order of the walk when calculating the tour.
  - Now that we have added the node at **minIndex** to the tree, some of the nodes not yet in the tree may be connected to the tree by edges with weight smaller than the values stored in their Data. These edges will all come from the node at **minIndex**. We therefore need to examine all nodes adjacent to this node. For each adjacent node not in the tree, if the edge from **minIndex** has smaller weight than this node's Data, its Data needs to be updated to the weight of the edge. In this case, the node's parent also needs to be updated to **minIndex**.
- Return the root of the **tree.**

UserInterface

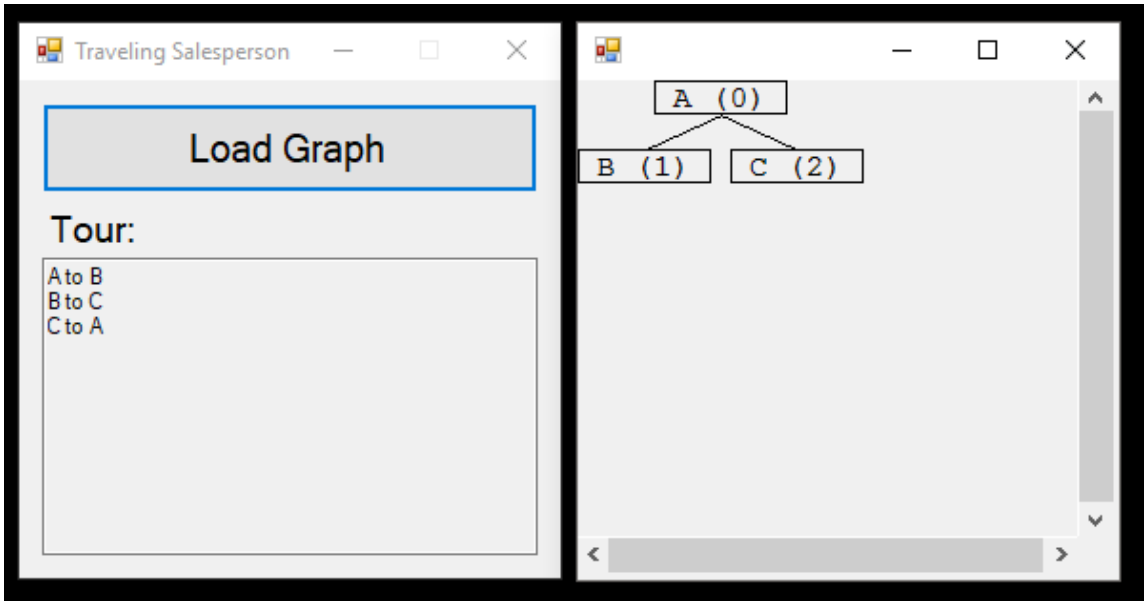
This is the class behind the user interface. It should have the following methods:

- **public UserInterface():**
  - Default constructor for the class.
- **private UndirectedGraph ReadGraph(string fileName):**
  - This method should create and return an **UndirectedGraph** from the given graph text file using a **StreamReader**. Use the information in the file as described in the previously mentioned File Format section to create the graph and add all of the edges.
- **private void uxOpenButton_Click(object sender, EventArgs e):**
  - This is a click event handler for the Load Graph button. This method should load the graph selected by the user through the **uxOpenDialog** and create a MST. Then the method should display the tree using **TreeForm** from the included **TreeViewer2** library and display the approximation (tour) to the solution of the traveling salesperson problem for the graph in the **uxTour** TextBox.This function should catch any exception thrown and display it using a **MessageBox**.

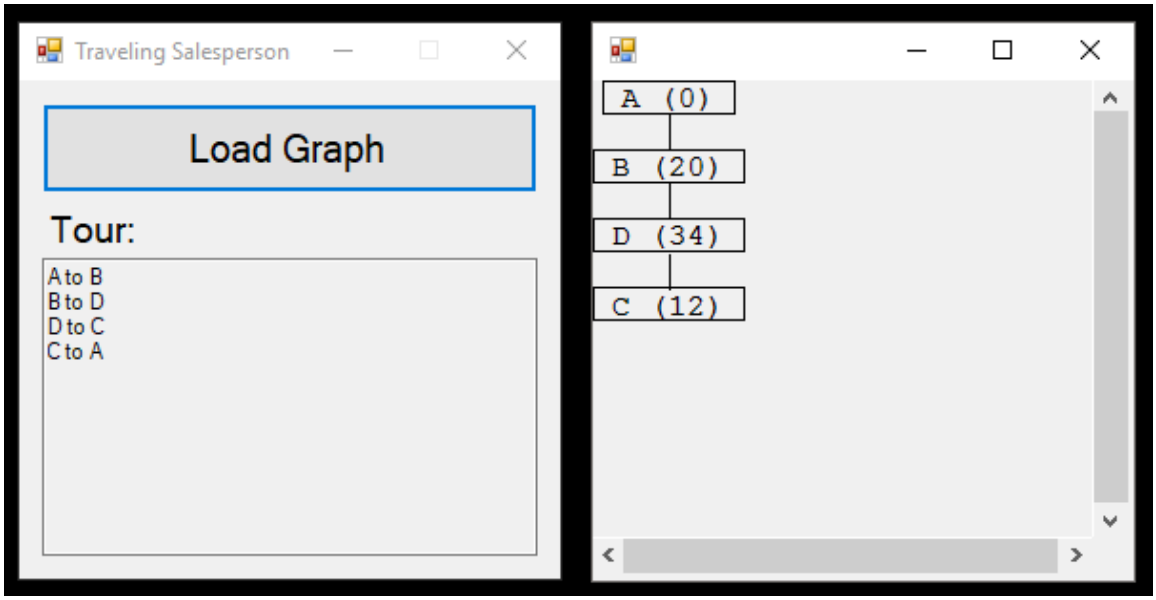
Testing

Included in the **Data** folder of the repository, there are a series of text files that contain graph data. You should get the following traveling salesperson tour and MST from each one:

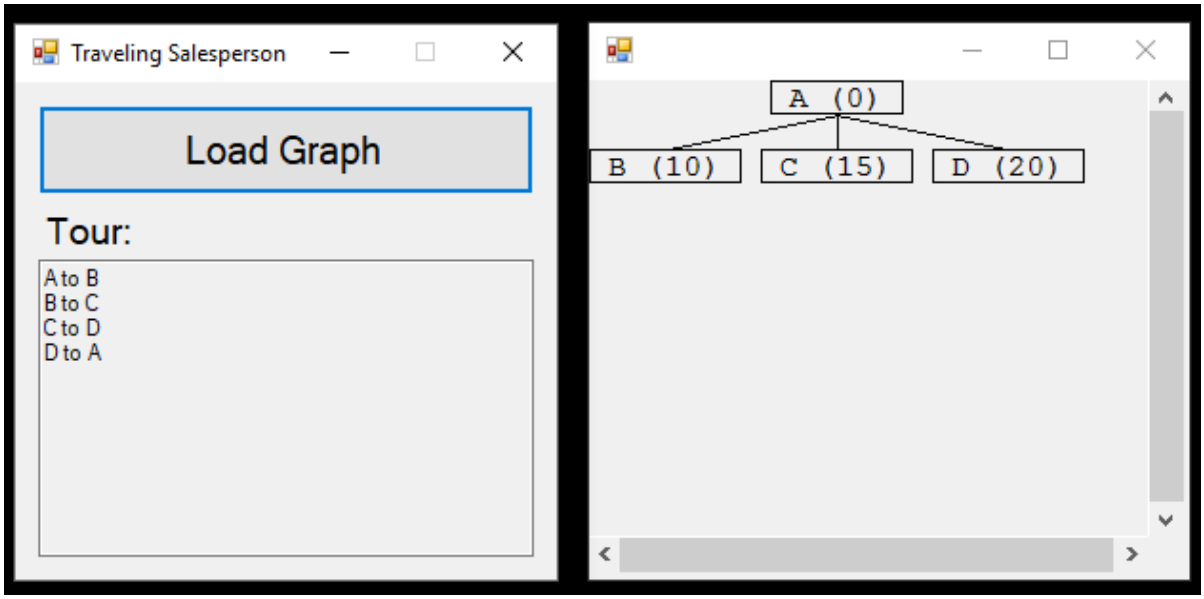
test1.txt



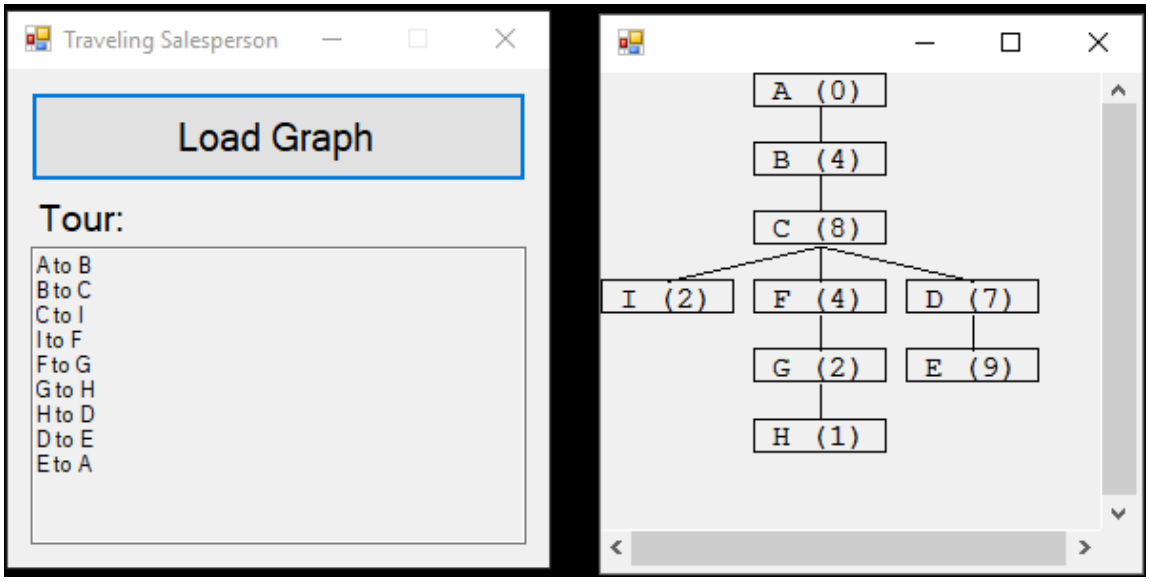
test2.txt



test3.txt



test4.txt



test5.txt and test6.txt

Test 5 should show an InvalidOperation exception. Test 6 will throw an ArgumentOutOfRange exception.

Submitting Your Assignment

Be sure you **commit** all your changes and **push** your commits to your GitHub repository. (**Note:** Prior to committing changes, it is a good idea - particularly in VS 2017 - to refresh the Team Explorer by clicking the circular arrow icon at the top. This seems to avoid many of the problems VS 2017 has with source control.) Then submit the *entire URL* of the **commit** that you want graded. There is no need to submit a comment, as you will not have a completion code.

Important: If the URL you submit does not contain the 40-hex-digit fingerprint of the commit you want graded, **you will receive a 0**, as this fingerprint is the only way we can verify that you completed your code prior to submitting your assignment. We will only grade the source code that is included in the commit that

you submit. Therefore, be sure that the commit on GitHub contains all four ".cs" files, and that they are the version you want graded. This is especially important if you had any trouble committing or pushing your code.

A Note on the Executable Provided

The provided executable has been *obfuscated* to prevent decompiling to the original source code. Code decompiled from obfuscated code is difficult or impossible to understand. There exist deobfuscators that can make this decompiled code a bit more understandable, but it's still fairly obvious that it is obfuscated code. **Any use of code decompiled and/or deobfuscated from the provided executable will be considered as cheating.**

Homework 4 Grading Criteria		
Criteria	**Ratings**	**Pts**
GUI Design		15.0 pts
Coding and Style Requirements		25.0 pts
Functionality		50.0 pts
Performance		10.0 pts
Late Penalty		0.0 pts
		Total Points: 100.0