Prim's Algorithm

Joshua Rand

CS312

Purpose

The purpose of this project was to create a visual windows program to implement Prim's Algorithm and manual-entry edge connection on a graph.

Summary

Prim's algorithm is one of many algorithms used in graph data structures, as it is used to construct a minimum spanning tree. It involves the selection of any arbitrary node or "vertex" in the graph and adding it to a tree. The program searches all edges connected to that vertex that do not connect to vertices in the tree and selects the connection with the smallest weight. The program then selects the vertex that the edge connects to and adds it to the tree with the starting vertex. The algorithm repeats, searching the edges connected to all of the tree nodes, until all of the nodes in the graph are in the tree.

Conclusion

I was able to create a visual representation of a graph data structure by drawing various graphics on a Windows Form panel. I used text labels to represent nodes and edge weights, and lines to represent edges. The labels that represent nodes appear in front of a light blue square and edges are drawn with light blue lines. Labels and edges that are in the spanning tree are colored dark blue. In the program, information about the graph, its nodes, edges, edge weights, and minimal spanning tree are all stored in a Graph class. In the project directory, there is a text document that contains several lines with 3 numbers on each line. Each line represents an edge in the graph, with the first and second numbers being the indices of the connected nodes, and the third number being the weight of the edge. When the program starts, it loads strings from the text document, converts them to integers, and inserts them into the graph by calling its Connect() function for each set of 3 integers. The program then draws all of the graph nodes, scattered in arbitrary locations, and the connecting edges along with their weights. By using the text boxes and "Connect" button provided, the user can manually add two edges to a spanning tree. The user is also provided another text box and a "Prim" button so they can specify a starting node and have the program run Prim's Algorithm to create a spanning tree.

Structure

I used a couple of classes to implement Prim's algorithm. One of them is a struct called Connection, which I used to keep track of all edges in the Graph.

Connection.h

```
#pragma once
struct Connection {
	int Node1;
	int Node2;
	int Weight;
};
```

Each Connection indicates the two nodes that it connects, and its weight. New Connections are created when vertices and edges are loaded into the Graph at the start of the program. The Connections are then added to the private array in the Graph class.

Graph.h

```
Connection connectionList[100];
```

In addition to a Connection array, I also implemented a 2x2 int array in the Graph class to keep track of edges and weights. Because there are 16 vertices in the provided text document, the array is 16x16. The array is implemented dynamically with pointers, so it is technically an array of arrays. The array contains 16 values, and each of those values is an array of 16 integers.

Graph.h

```
int **data;
```

Graph.cpp

```
data = new int*[width];
for (int i = 0; i < height; i++) {
	data[i] = new int[height];
}
for (int i = 0; i < width; i++) {
	for (int j = 0; j < height; j++) {
		data[i][j] = 0;
	}
}
```

The main reason I used a Connection array in addition to a 2x2 int array, is so that I could keep track of all unique edges without having duplicates. In order to draw the edges, one might scan each vertex and draw all the edges connecting to it, but if the program scans two vertices that are connected to each other, each edge will be drawn twice.

Setup

The program starts by initializing a Graph object and calling the local LoadGraph() function. The LoadGraph() function loads the text document as an input file stream (ifstream), parses the strings to integers, and loads them into the Graph.

MyForm.h

```
void LoadGraph(Graph *g) {
	ifstream file;
	file.open("Small Graph.txt");
	if (file.fail()) {
		Application::Exit();
	}
	while (!file.eof()) {
		string n1, n2, w;
		file >> n1 >> n2 >> w;
		if (n1 != "" && n1 != "Vertex")
		{
			bool result = g->Connect(stoi(n1), stoi(n2), stoi(w));
			if (!result) {

			}
		}
	}
	file.close();
}
```

The Connect() function creates a Connection and adds it to the Graph's Connection array. It also adds the weight of the edge in the 2x2 int array in two places. Node1 represents distance along one axis while Node2 represents distance along the other. See sample output below for an example.

Graph.cpp

```
bool Graph::Connect(int node1, int node2, int weight) {
	if (data[node1][node2] != 0 || data[node2][node1] != 0)
		return false;
	data[node1][node2] = weight;
	data[node2][node1] = weight;
	Connection c;
	c.Node1 = node1;
	c.Node2 = node2;
	c.Weight = weight;
	//connectionList.Add(*c);
	connectionList[connectionCount] = c;

	//if (!tree[node1])
	//	tree[node1] = true;
	//if (!tree[node2])
	//	tree[node2] = true;

	connectionCount++;
	return true;
}
```

After calling the Connect() function for each of the edges in the Graph, the graph then creates labels for each vertex and edge using the local newLabel() and newLabel2() functions. newLabel() creates a new text label with black text and a blue background, while newLabel2() creates a new text label with smaller black text and a transparent background. newLabel() is used for creating vertices, while newLabel2() is used for creating weight numbers for each edge.

MyForm.h

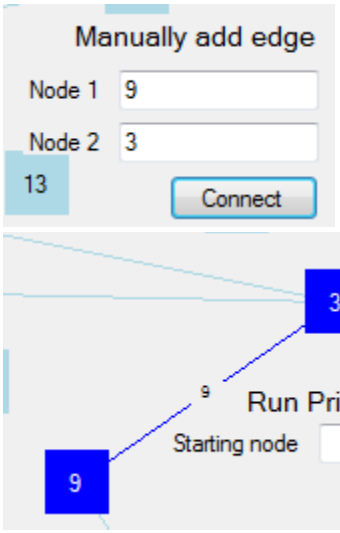
```
Label^ newLabel(System::String ^text, int x, int y){
	Label^ l = gcnew Label();
	//l->AutoSize = true;
	l->BackColor = System::Drawing::Color::LightBlue;
	l->Location = System::Drawing::Point(x, y);
	l->Name = L"label1asdf";
	l->Size = System::Drawing::Size(32, 32);
	l->TabIndex = 0;
	l->Text = text;
	l->TextAlign = System::Drawing::ContentAlignment::MiddleCenter;
	this->panel1->Controls->Add(l);
	return l;
}

Label^ newLabel2(System::String^ text, int x, int y){
	Label^ l = gcnew Label();
	//l->AutoSize = true;
	l->BackColor = System::Drawing::Color::Transparent;
	l->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 6,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
		static_cast<System::Byte>(0)));
	l->Location = System::Drawing::Point(x, y);
	l->Name = L"label1asdf";
	l->Size = System::Drawing::Size(16, 16);
	l->TabIndex = 0;
	l->Text = text;
	l->TextAlign = System::Drawing::ContentAlignment::MiddleCenter;
	this->panel1->Controls->Add(l);
	return l;
}
```

The panel1_Paint() function runs after the program starts and anytime the panel is refreshed. In the function is a for loop that iterates through all Connections in the Graph and draws lines that connect the corresponding vertices together.

```
for (int i = 0; i < graph->ConnectionCount(); i++) {
	Connection *c = graph->GetConnection(i);
	Label ^s = nodes[c->Node1];
	Label ^e = nodes[c->Node2];
	int weight = c->Weight;
	if (graph->IsPrim(c->Node1, c->Node2))
		g->DrawLine(spanPen, s->Location.X + 16, s->Location.Y + 16, e->Location.X
+ 16, e->Location.Y + 16);
	else
		g->DrawLine(normalPen, s->Location.X + 16, s->Location.Y + 16, e-
>Location.X + 16, e->Location.Y + 16);
}
```

Manual Connection

The user has the option to manually enter 2 vertices in the graph and add them to the spanning tree.



Clicking on the Connection button calls the local connectbutton_Click() method, which retrieves the text from the two node text boxes, converts them to integers, then passes calls the local TryPrim() method.

MyForm.h

```
private: System::Void connectbutton_Click(System::Object^  sender, System::EventArgs^  e)
{
	//int node1 = Convert::ToInt32(node1box->Text);
	//int node2 = Convert::ToInt32(node2box->Text);
	//int weight = Convert::ToInt32(weightbox->Text);
	int node1 = (node1box->Text == "") ? -1 : int::Parse(node1box->Text);
	int node2 = (node2box->Text == "") ? -1 : int::Parse(node2box->Text);
	//int weight = (weightbox->Text == "") ? -1 : int::Parse(weightbox->Text);

	if (node1 == -1 || node2 == -1) {
		errorlabel->Text = "Error: Node index must not be empty";
		panel1->Refresh();
		return;
	}

	String ^message = gcnew String("");
	bool success = TryPrim(node1, node2, 5, message);
	if (!success) {
		errorlabel->Text = "Error: " + message;
	}
	else {
		errorlabel->Text = "";
		panel1->Refresh();
	}
}
```

The local TryPrim() method does input verification (make sure node indices are within the array bounds, prevent the same nodes from being connected, make sure the nodes are actually connected in the graph, make sure the nodes are not already both in the spanning tree.

MyForm.h

```
bool TryPrim(int node1, int node2, int weight, String ^&message) {
	if (node1 < 0 || node1 > ARRAY_SIZE - 1) {
		message = "Invalid index for node 1";
		return false;
	}
	else if (node2 < 0 || node2 > ARRAY_SIZE - 1) {
		message = "Invalid index for node 2";
		return false;
	}
	else if (node1 == node2) {
		message = "Cannot connect to the same node";
		return false;
	}
	else if (!graph->ConnectionExists(node1, node2)) {
		message = "These nodes are not connected";
		return false;
	}
	else if (graph->IsPrim(node1, node2)) {
		message = "Prim connection already exists";
		return false;
	}
	else if (weight <= 0) {
		message = "Weight cannot be 0";
		return false;
	}
	graph->PrimEdge(node1, node2);
	message = "Success";

	SetBlueLabel(nodes[node1]);
	SetBlueLabel(nodes[node2]);
	return true;
}
```

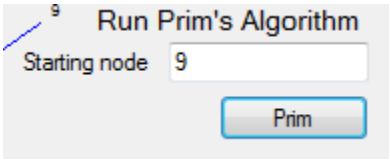
The local SetBlueLabel() method changes the colors of the nodes to a darker shade of blue. The Graph's PrimEdge() method adds the two vertices into the Graph's spanning tree.

Graph.cpp

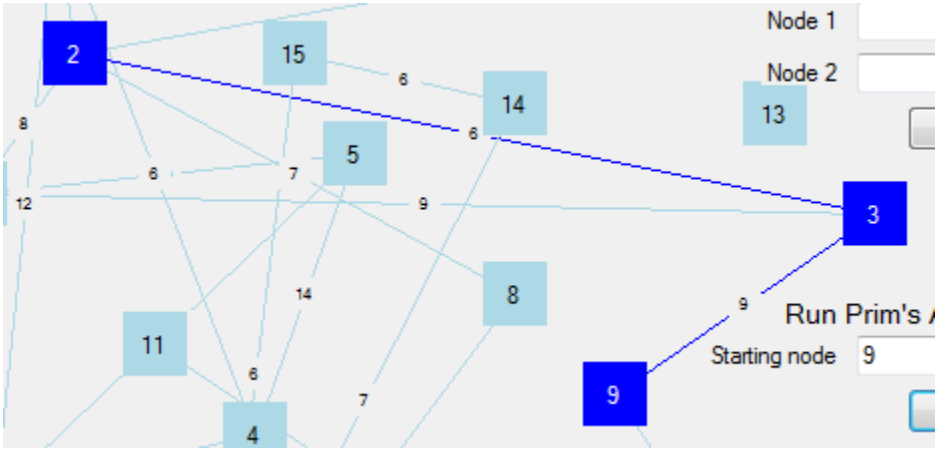
```
bool Graph::PrimEdge(int node1, int node2) {

	if (data[node1][node2] == 0 || data[node2][node1] == 0) {
		return false;
	}

	prim[node1][node2] = true;
	prim[node2][node1] = true;

	if (!tree[node1])
		tree[node1] = true;
	if (!tree[node2])
		tree[node2] = true;
	return true;
}
```

Prim's Algorithm

The user also has the option to run Prim's Algorithm by specifying a starting vertex and then repeatedly clicking on the "Prim" button.



The result of clicking the Prim button twice:



Each click of the Prim button calls the local PrimButton_Click() function,

MyForm.h

```
private: System::Void PrimButton_Click(System::Object^  sender, System::EventArgs^  e) {
	int vertex = (StartVertex->Text == "") ? -1 : int::Parse(StartVertex->Text);
	int blueNode = -1;
	if (!graph->InTree(vertex))
		SetBlueLabel(nodes[vertex]);
	graph->Prim(vertex, blueNode);
	if (blueNode != -1)
		SetBlueLabel(nodes[blueNode]);
	panel1->Refresh();
}
```

The local PrimButton_Click() function calls the Graph's Prim() function, which searches for the edge connecting to the tree with the smallest weight. It then adds the connected vertex to the graph by calling the PrimEdge() function. The parameters it uses are two nodes that are connected by the edge.

Graph.cpp

```
void Graph::Prim(int startVertex, int &blueVertex) {
	if (IsTreeComplete())
		return;
	//Starting the tree
	if (!tree[startVertex]) {
		tree[startVertex] = true;
	}
	int node1;
	int node2 = GetClosestNode(node1);

	if (node2 != -1) {
		PrimEdge(node1, node2);
		blueVertex = node2;
	}
}
```

The GetClosestNode() method iterates through all vertices that are in the tree and finds an edge connected to the tree with the smallest weight. The vertex in the tree connected to the edge is set to the node1 reference parameter, and the function returns the vertex connected to the edge that is not in the tree.

```
int Graph::GetClosestNode(int &node1) {
	int weight = 99999;
	int closestNode = -1;
	for (int node = 0; node < width; node++) {
		if (tree[node]) {
			for (int i = 0; i < width; i++) {
				if (node != i && data[node][i] < weight && data[node][i] != 0
&& !IsPrim(node, i) && !tree[i]) {
					weight = data[node][i];
					closestNode = i;
					node1 = node;
				}
			}
		}
	}
	return closestNode;
}
```

Conclusion

I have known about Prim's Algorithm for quite a long time, and it was very interesting to be able to implement it in a program. It is one of a few algorithms used to create minimum spanning trees. I was able to reduce complexity for counting and drawing the edges in the graph. Without the Connection array, it would have been $O(n^2)$, where n is the number of vertices, but now it is only $O(n^2)$ for the worst-case scenario, since the program only iterates through a list of edges once and not a list of vertices twice. In retrospect, I could have implemented the function for finding the closest vertex (on the edge with the smallest weight) a little differently. It may

have been more efficient to do a recursive breadth-first or depth-first search on each spanning tree node instead of iterating through every edge in the spanning tree.

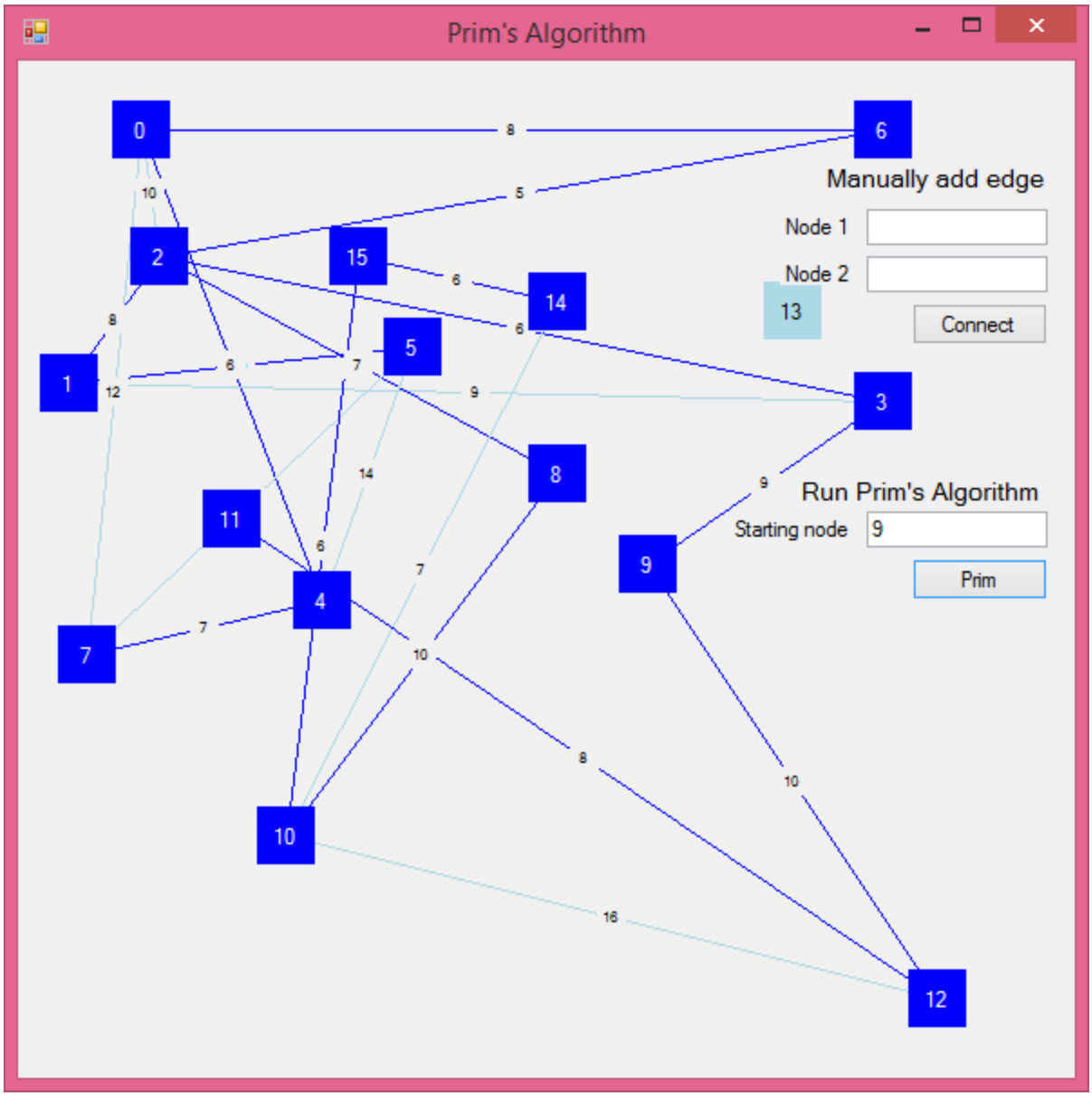
Input text file

Vertex	edge	weight
0	2	10
0	6	8
0	4	6
0	7	12
1	2	8
1	3	9
1	5	10
2	3	6
2	1	8
2	0	10
2	6	5
2	8	7
3	1	9
3	2	6
3	9	9
4	5	14
4	7	7
4	0	6
5	4	14
5	7	16
5	1	10
6	2	5
6	0	8
7	5	16
7	0	12
7	4	7
8	2	7
8	10	10

9	3	9
9	12	10
10	8	10
10	12	16
10	14	7
10	15	6
11	12	8
12	11	8
12	9	10
12	10	16
14	10	7
14	15	6
15	10	9
15	14	6

Sample Run

(Input 9 and repeatedly press Prim)



Connection Matrix

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	10	0	6	0	8	12	0	0	0	0	0	0	0	0
1	0	0	8	9	0	10	0	0	0	0	0	0	0	0	0	0
2	10	8	0	6	0	0	5	0	7	0	0	0	0	0	0	0
3	0	9	6	0	0	0	0	0	0	9	0	0	0	0	0	0
4	6	0	0	0	0	14	0	7	0	0	0	0	0	0	0	0
5	0	10	0	0	14	0	0	16	0	0	0	0	0	0	0	0
6	8	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0
7	12	0	0	0	7	16	0	0	0	0	0	0	0	0	0	0
8	0	0	7	0	0	0	0	0	0	0	10	0	0	0	0	0
9	0	0	0	9	0	0	0	0	0	0	0	0	10	0	0	0

10	0	0	0	0	0	0	0	0	10	0	0	0	16	0	7	6
11	0	0	0	0	0	0	0	0	0	0	0	0	8	0	0	0
12	0	0	0	0	0	0	0	0	0	10	16	8	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	7	0	0	0	0	6
15	0	0	0	0	0	0	0	0	0	0	6	0	0	0	6	0

Spanning Tree Matrix

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
2	0	1	0	1	0	0	1	0	1	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0
4	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
5	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
8	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
10	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
11	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
12	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
15	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0

Source Code

Connection.h

```
#pragma once
struct Connection {
	int Node1;
	int Node2;
	int Weight;
};
```

Graph.h

```
#pragma once
#include "Connection.h"
#include "PtrArray.h"
using namespace std;
using namespace System::Collections::Generic;

class Graph {
public:
	Graph(int, int);
	~Graph();
	//void Display() const;
	bool Connect(int, int, int);
	void Prim(int, int &);
	bool PrimEdge(int, int);
	bool ConnectionExists(int, int) const;
	bool IsPrim(int, int) const;
	bool InTree(int) const;
	//void SetBeginning();
	//bool AtEnd() const;
	Connection *GetConnection(int);
	int ConnectionCount() const;
private:
	Connection *GetConnection(int, int);
	int GetClosestNode(int &);
	bool IsTreeComplete();
	//PtrArray<Connection> connectionList;
	Connection connectionList[100];
	//list<Connection> connectionList;
	//list<Connection>::iterator it;
	bool *tree;
	bool **prim;
	int **data;
	int width, height;
	int connectionCount;
};
```

Graph.cpp

```
#include "Graph.h"

Graph::Graph(int width, int height) {
	connectionCount = 0;
	this->width = width;
	this->height = height;
	tree = new bool[width];
	prim = new bool*[width];
	for (int i = 0; i < width; i++) {
		tree[i] = false;
	}
	data = new int*[width];
	for (int i = 0; i < height; i++) {
		data[i] = new int[height];
	}
	for (int i = 0; i < width; i++) {
		for (int j = 0; j < height; j++) {
			data[i][j] = 0;
		}
	}
	for (int i = 0; i < height; i++) {
		prim[i] = new bool[height];
	}
	for (int i = 0; i < width; i++) {
		for (int j = 0; j < height; j++) {
			prim[i][j] = false;
		}
	}
}

Graph::~Graph() {
	for (int i = 0; i < height; i++) {
		delete[] data[i];
	}
	delete[] data;
	delete[] tree;
	for (int i = 0; i < height; i++) {
		delete[] prim[i];
	}
	delete[] prim;
}

//void Matrix::Display() const {
//	cout << "   ";
//	for (int i = 0; i < width; i++) {
//		cout << i << " " << ((i > 9) ? "" : " ");
//	}
//	cout << endl;
//
//	for (int i = 0; i < height; i++) {
//		cout << i << " " << ((i > 9) ? "" : " ");
//		for (int j = 0; j < width; j++) {
//			cout << data[j][i] << " " << ((data[j][i] > 9) ? "" : " ");
//		}
//		cout << std::endl;
//	}
```

```
//}

bool Graph::Connect(int node1, int node2, int weight) {
	if (data[node1][node2] != 0 || data[node2][node1] != 0)
		return false;
	data[node1][node2] = weight;
	data[node2][node1] = weight;
	Connection c;
	c.Node1 = node1;
	c.Node2 = node2;
	c.Weight = weight;
	//connectionList.Add(*c);
	connectionList[connectionCount] = c;

	//if (!tree[node1])
	//	tree[node1] = true;
	//if (!tree[node2])
	//	tree[node2] = true;

	connectionCount++;
	return true;
}

void Graph::Prim(int startVertex, int &blueVertex) {
	if (IsTreeComplete())
		return;
	//Starting the tree
	if (!tree[startVertex]) {
		tree[startVertex] = true;
	}
	int node1;
	int node2 = GetClosestNode(node1);

	if (node2 != -1) {
		PrimEdge(node1, node2);
		blueVertex = node2;
	}
}

bool Graph::IsTreeComplete() {
	bool complete = true;
	for (int i = 0; i < width; i++) {
		if (!tree[i])
			return false;
	}
	return true;
}

int Graph::GetClosestNode(int &node1) {
	int weight = 99999;
	int closestNode = -1;
	for (int node = 0; node < width; node++) {
		if (tree[node]) {
			for (int i = 0; i < width; i++) {
				if (node != i && data[node][i] < weight && data[node][i] != 0
&& !IsPrim(node, i) && !tree[i]) {
					weight = data[node][i];
					closestNode = i;
```

```
                                node1 = node;
                            }
                        }
                    }
        }
        return closestNode;
}

bool Graph::PrimEdge(int node1, int node2) {

        if (data[node1][node2] == 0 || data[node2][node1] == 0) {
                return false;
        }

        prim[node1][node2] = true;
        prim[node2][node1] = true;

        if (!tree[node1])
                tree[node1] = true;
        if (!tree[node2])
                tree[node2] = true;
        return true;
}

bool Graph::ConnectionExists(int node1, int node2) const {
        return (data[node1][node2] != 0 || data[node2][node1] != 0);
}

bool Graph::IsPrim(int node1, int node2) const {
        return (prim[node1][node2] || prim[node2][node1]);
}

bool Graph::InTree(int node) const {
        return (tree[node]);
}

//void Matrix::SetBeginning() {
//      it = connectionList.begin();
//}

//bool Matrix::AtEnd() const {
//      return (it == connectionList.end());
//}

//Connection Matrix::GetConnection() {
//      Connection c = *it;
//      ++it;
//      return c;
//}

Connection *Graph::GetConnection(int index) {
        return &connectionList[index];
}

Connection *Graph::GetConnection(int node1, int node2) {
        for (int i = 0; i < connectionCount; i++) {
                if (connectionList[i].Node1 == node1 && connectionList[i].Node2 == node2) {
                        return &connectionList[i];
```

```
            }
        }

        return nullptr;
}

int Graph::ConnectionCount() const {
        return connectionCount;
}
```

MyForm.h

```
#pragma once
#include "Graph.h"
#include <fstream>
#include <string>

const int ARRAY_SIZE = 16;

namespace Prim {
        using namespace System;
        using namespace System::ComponentModel;
        using namespace System::Collections;
        using namespace System::Windows::Forms;
        using namespace System::Data;
        using namespace System::Drawing;

        /// <summary>
        /// Summary for MyForm
        /// </summary>
        public ref class MyForm : public System::Windows::Forms::Form
        {
        public:
                MyForm(void)
                {
                        InitializeComponent();
                        //
                        //TODO: Add the constructor code here
                        //
                }

        protected:
                /// <summary>
                /// Clean up any resources being used.
                /// </summary>
                ~MyForm()
                {
                        if (components)
                        {
                                delete components;
                        }
                        delete graph;
                }
        private: System::Windows::Forms::Button^  connectbutton;
        protected:

        private: System::Windows::Forms::TextBox^  node2box;
        private: System::Windows::Forms::TextBox^  node1box;
```

```
private: System::Windows::Forms::Label^  errorlabel;

private: System::Windows::Forms::Label^  label2;
private: System::Windows::Forms::Label^  label1;
private: System::Windows::Forms::Label^  label4;
private: System::Windows::Forms::Label^  label5;
private: System::Windows::Forms::TextBox^  StartVertex;
private: System::Windows::Forms::Button^  PrimButton;
private: System::Windows::Forms::Label^  label3;



protected:
private: System::Windows::Forms::Panel^  panel1;



protected:
private:

	void LoadGraph(Graph *g) {
		ifstream file;
		file.open("Small Graph.txt");
		if (file.fail()) {
			Application::Exit();
		}
		while (!file.eof()) {
			string n1, n2, w;
			file >> n1 >> n2 >> w;
			if (n1 != "" && n1 != "Vertex")
			{
				bool result = g->Connect(stoi(n1), stoi(n2), stoi(w));
				if (!result) {

				}
			}
		}
		file.close();
	}

	bool TryPrim(int node1, int node2, int weight, String ^&message) {
		if (node1 < 0 || node1 > ARRAY_SIZE - 1) {
			message = "Invalid index for node 1";
			return false;
		}
		else if (node2 < 0 || node2 > ARRAY_SIZE - 1) {
			message = "Invalid index for node 2";
			return false;
		}
		else if (node1 == node2) {
			message = "Cannot connect to the same node";
			return false;
		}
		else if (!graph->ConnectionExists(node1, node2)) {
			message = "These nodes are not connected";
			return false;
		}
```

```
			else if (graph->IsPrim(node1, node2)) {
				message = "Prim connection already exists";
				return false;
			}
			else if (weight <= 0) {
				message = "Weight cannot be 0";
				return false;
			}
			graph->PrimEdge(node1, node2);
			message = "Success";

			SetBlueLabel(nodes[node1]);
			SetBlueLabel(nodes[node2]);
			return true;
		}

		Label^ newLabel(System::String ^text, int x, int y){
			Label^ l = gcnew Label();
			//l->AutoSize = true;
			l->BackColor = System::Drawing::Color::LightBlue;
			l->Location = System::Drawing::Point(x, y);
			l->Name = L"label1asdf";
			l->Size = System::Drawing::Size(32, 32);
			l->TabIndex = 0;
			l->Text = text;
			l->TextAlign = System::Drawing::ContentAlignment::MiddleCenter;
			this->panel1->Controls->Add(l);
			return l;
		}

		Label^ newLabel2(System::String^ text, int x, int y){
			Label^ l = gcnew Label();
			//l->AutoSize = true;
			l->BackColor = System::Drawing::Color::Transparent;
			l->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 6,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
				static_cast<System::Byte>(0)));
			l->Location = System::Drawing::Point(x, y);
			l->Name = L"label1asdf";
			l->Size = System::Drawing::Size(16, 16);
			l->TabIndex = 0;
			l->Text = text;
			l->TextAlign = System::Drawing::ContentAlignment::MiddleCenter;
			this->panel1->Controls->Add(l);
			return l;
		}

		void SetBlueLabel(Label ^label) {
			label->BackColor = System::Drawing::Color::Blue;
			label->ForeColor = System::Drawing::Color::White;
		}
		/// <summary>
		/// Required designer variable.
		/// </summary>
		System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
		/// <summary>
```

```
		/// Required method for Designer support - do not modify
		/// the contents of this method with the code editor.
		/// </summary>
		void InitializeComponent(void)
		{
			this->panel1 = (gcnew System::Windows::Forms::Panel());
			this->label3 = (gcnew System::Windows::Forms::Label());
			this->PrimButton = (gcnew System::Windows::Forms::Button());
			this->label5 = (gcnew System::Windows::Forms::Label());
			this->StartVertex = (gcnew System::Windows::Forms::TextBox());
			this->label4 = (gcnew System::Windows::Forms::Label());
			this->label2 = (gcnew System::Windows::Forms::Label());
			this->label1 = (gcnew System::Windows::Forms::Label());
			this->errorlabel = (gcnew System::Windows::Forms::Label());
			this->node2box = (gcnew System::Windows::Forms::TextBox());
			this->node1box = (gcnew System::Windows::Forms::TextBox());
			this->connectbutton = (gcnew System::Windows::Forms::Button());
			this->panel1->SuspendLayout();
			this->SuspendLayout();
			//
			// panel1
			//
			this->panel1->Controls->Add(this->label3);
			this->panel1->Controls->Add(this->PrimButton);
			this->panel1->Controls->Add(this->label5);
			this->panel1->Controls->Add(this->StartVertex);
			this->panel1->Controls->Add(this->label4);
			this->panel1->Controls->Add(this->label2);
			this->panel1->Controls->Add(this->label1);
			this->panel1->Controls->Add(this->errorlabel);
			this->panel1->Controls->Add(this->node2box);
			this->panel1->Controls->Add(this->node1box);
			this->panel1->Controls->Add(this->connectbutton);
			this->panel1->Location = System::Drawing::Point(12, 12);
			this->panel1->Name = L"panel1";
			this->panel1->Size = System::Drawing::Size(560, 538);
			this->panel1->TabIndex = 0;
			this->panel1->Paint += gcnew
System::Windows::Forms::PaintEventHandler(this, &MyForm::panel1_Paint);
			//
			// label3
			//
			this->label3->AutoSize = true;
			this->label3->Location = System::Drawing::Point(381, 240);
			this->label3->Name = L"label3";
			this->label3->Size = System::Drawing::Size(70, 13);
			this->label3->TabIndex = 11;
			this->label3->Text = L"Starting node";
			//
			// PrimButton
			//
			this->PrimButton->Location = System::Drawing::Point(482, 263);
			this->PrimButton->Name = L"PrimButton";
			this->PrimButton->Size = System::Drawing::Size(75, 23);
			this->PrimButton->TabIndex = 10;
			this->PrimButton->Text = L"Prim";
			this->PrimButton->UseVisualStyleBackColor = true;
```

```
                    this->PrimButton->Click += gcnew System::EventHandler(this,
&MyForm::PrimButton_Click);
                    //
                    // label5
                    //
                    this->label5->AutoSize = true;
                    this->label5->Font = (gcnew System::Drawing::Font(L"Microsoft Sans
Serif", 10, System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                           static_cast<System::Byte>(0)));
                    this->label5->Location = System::Drawing::Point(418, 217);
                    this->label5->Name = L"label5";
                    this->label5->Size = System::Drawing::Size(139, 17);
                    this->label5->TabIndex = 9;
                    this->label5->Text = L"Run Prim\'s Algorithm";
                    //
                    // StartVertex
                    //
                    this->StartVertex->Location = System::Drawing::Point(457, 237);
                    this->StartVertex->Name = L"StartVertex";
                    this->StartVertex->Size = System::Drawing::Size(100, 20);
                    this->StartVertex->TabIndex = 8;
                    //
                    // label4
                    //
                    this->label4->AutoSize = true;
                    this->label4->Font = (gcnew System::Drawing::Font(L"Microsoft Sans
Serif", 10, System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                           static_cast<System::Byte>(0)));
                    this->label4->Location = System::Drawing::Point(432, 44);
                    this->label4->Name = L"label4";
                    this->label4->Size = System::Drawing::Size(128, 17);
                    this->label4->TabIndex = 7;
                    this->label4->Text = L"Manually add edge";
                    //
                    // label2
                    //
                    this->label2->AutoSize = true;
                    this->label2->Location = System::Drawing::Point(409, 99);
                    this->label2->Name = L"label2";
                    this->label2->Size = System::Drawing::Size(42, 13);
                    this->label2->TabIndex = 5;
                    this->label2->Text = L"Node 2";
                    //
                    // label1
                    //
                    this->label1->AutoSize = true;
                    this->label1->Location = System::Drawing::Point(409, 73);
                    this->label1->Name = L"label1";
                    this->label1->Size = System::Drawing::Size(42, 13);
                    this->label1->TabIndex = 5;
                    this->label1->Text = L"Node 1";
                    //
                    // errorlabel
                    //
                    this->errorlabel->AutoSize = true;
                    this->errorlabel->Location = System::Drawing::Point(231, 13);
                    this->errorlabel->Name = L"errorlabel";
                    this->errorlabel->Size = System::Drawing::Size(28, 13);
```

```
                    this->errorlabel->TabIndex = 4;
                    this->errorlabel->Text = L"error";
                    this->errorlabel->TextAlign =
System::Drawing::ContentAlignment::TopRight;
                    //
                    // node2box
                    //
                    this->node2box->Location = System::Drawing::Point(457, 96);
                    this->node2box->Name = L"node2box";
                    this->node2box->Size = System::Drawing::Size(100, 20);
                    this->node2box->TabIndex = 2;
                    //
                    // node1box
                    //
                    this->node1box->Location = System::Drawing::Point(457, 70);
                    this->node1box->Name = L"node1box";
                    this->node1box->Size = System::Drawing::Size(100, 20);
                    this->node1box->TabIndex = 1;
                    //
                    // connectbutton
                    //
                    this->connectbutton->Location = System::Drawing::Point(482, 122);
                    this->connectbutton->Name = L"connectbutton";
                    this->connectbutton->Size = System::Drawing::Size(75, 23);
                    this->connectbutton->TabIndex = 0;
                    this->connectbutton->Text = L"Connect";
                    this->connectbutton->UseVisualStyleBackColor = true;
                    this->connectbutton->Click += gcnew System::EventHandler(this,
&MyForm::connectbutton_Click);
                    //
                    // MyForm
                    //
                    this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
                    this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
                    this->ClientSize = System::Drawing::Size(584, 562);
                    this->Controls->Add(this->panel1);
                    this->Name = L"MyForm";
                    this->Text = L"Prim\'s Algorithm";
                    this->Load += gcnew System::EventHandler(this,
&MyForm::MyForm_Load);
                    this->panel1->ResumeLayout(false);
                    this->PerformLayout();
                    this->ResumeLayout(false);

                }
#pragma endregion

                //array<Rectangle*>^ nodes;
                cli::array<Label^>^ nodes;
                Graphics ^g;
                SolidBrush ^b;
                Pen ^normalPen;
                Pen ^spanPen;
                Graph *graph = nullptr;

        private: System::Void MyForm_Load(System::Object^  sender, System::EventArgs^  e)
{
                errorlabel->Text = "";
```

```
            graph = new Graph(ARRAY_SIZE, ARRAY_SIZE);
            LoadGraph(graph);
            //nodes = gcnew array<Rectangle*>(ARRAY_SIZE);
            nodes = gcnew cli::array<Label^, 1>(ARRAY_SIZE);
            g = panel1->CreateGraphics();
            b = gcnew SolidBrush(Color::Blue);
            normalPen = gcnew Pen(Color::LightBlue);
            spanPen = gcnew Pen(Color::Blue);

            //Scatter the nodes in random places around the panel
            nodes[0] = newLabel("0", 40, 10);
            nodes[1] = newLabel("1", 0, 150);
            nodes[2] = newLabel("2", 50, 80);
            nodes[3] = newLabel("3", 450, 160);
            nodes[4] = newLabel("4", 140, 270);
            nodes[5] = newLabel("5", 190, 130);
            nodes[6] = newLabel("6", 450, 10);
            nodes[7] = newLabel("7", 10, 300);
            nodes[8] = newLabel("8", 270, 200);
            nodes[9] = newLabel("9", 320, 250);
            nodes[10] = newLabel("10", 120, 400);
            nodes[11] = newLabel("11", 90, 225);
            nodes[12] = newLabel("12", 480, 490);
            nodes[13] = newLabel("13", 400, 110);
            nodes[14] = newLabel("14", 270, 105);
            nodes[15] = newLabel("15", 160, 80);

            //Add weights
            for (int i = 0; i < graph->ConnectionCount(); i++){
                  Connection *c = graph->GetConnection(i);
                  Point node1 = nodes[c->Node1]->Location;
                  Point node2 = nodes[c->Node2]->Location;
                  Point center = Point(node1.X + (node2.X - node1.X) / 2, node1.Y +
(node2.Y - node1.Y) / 2);
                  newLabel2(c->Weight.ToString(), center.X + 8, center.Y + 8);
            }

      }
      private: System::Void panel1_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e) {
            //g->FillEllipse(b, 0, 0, 10, 10);
            //for (int i = 0; i < ARRAY_SIZE; i++) {
            //    for (int j = 0; j < ARRAY_SIZE; j++){
            //          //if (connectionMatrix[i][j] > 0){
            //          //    Label ^s = nodes[i];
            //          //    Label ^e = nodes[j];
            //          //    g->DrawLine(normalPen, s->Location.X + 16, s-
>Location.Y + 16, e->Location.X + 16, e->Location.Y + 16);
            //          //}
            //    }
            //}

            //Draw each connection
            //For loop currently causing issues
            for (int i = 0; i < graph->ConnectionCount(); i++) {
                  Connection *c = graph->GetConnection(i);
                  Label ^s = nodes[c->Node1];
                  Label ^e = nodes[c->Node2];
```

```
                        int weight = c->Weight;
                        if (graph->IsPrim(c->Node1, c->Node2))
                                g->DrawLine(spanPen, s->Location.X + 16, s->Location.Y + 16,
e->Location.X + 16, e->Location.Y + 16);
                        else
                                g->DrawLine(normalPen, s->Location.X + 16, s->Location.Y + 16,
e->Location.X + 16, e->Location.Y + 16);
                }
        }
        private: System::Void label2_Click(System::Object^  sender, System::EventArgs^  e)
{
        }
private: System::Void connectbutton_Click(System::Object^  sender, System::EventArgs^  e)
{
        //int node1 = Convert::ToInt32(node1box->Text);
        //int node2 = Convert::ToInt32(node2box->Text);
        //int weight = Convert::ToInt32(weightbox->Text);
        int node1 = (node1box->Text == "") ? -1 : int::Parse(node1box->Text);
        int node2 = (node2box->Text == "") ? -1 : int::Parse(node2box->Text);
        //int weight = (weightbox->Text == "") ? -1 : int::Parse(weightbox->Text);

        if (node1 == -1 || node2 == -1) {
                errorlabel->Text = "Error: Node index must not be empty";
                panel1->Refresh();
                return;
        }

        String ^message = gcnew String("");
        bool success = TryPrim(node1, node2, 5, message);
        if (!success) {
                errorlabel->Text = "Error: " + message;
        }
        else {
                errorlabel->Text = "";
                panel1->Refresh();
        }
}
private: System::Void PrimButton_Click(System::Object^  sender, System::EventArgs^  e) {
        int vertex = (StartVertex->Text == "") ? -1 : int::Parse(StartVertex->Text);
        int blueNode = -1;
        if (!graph->InTree(vertex))
                SetBlueLabel(nodes[vertex]);
        graph->Prim(vertex, blueNode);
        if (blueNode != -1)
                SetBlueLabel(nodes[blueNode]);
        panel1->Refresh();
}
};
}
```