Graph Data Structure

Joshua Rand

CS312

Purpose

The purpose of this project was to write a program to illustrate the graph data structure and perform several operations on it.

Summary

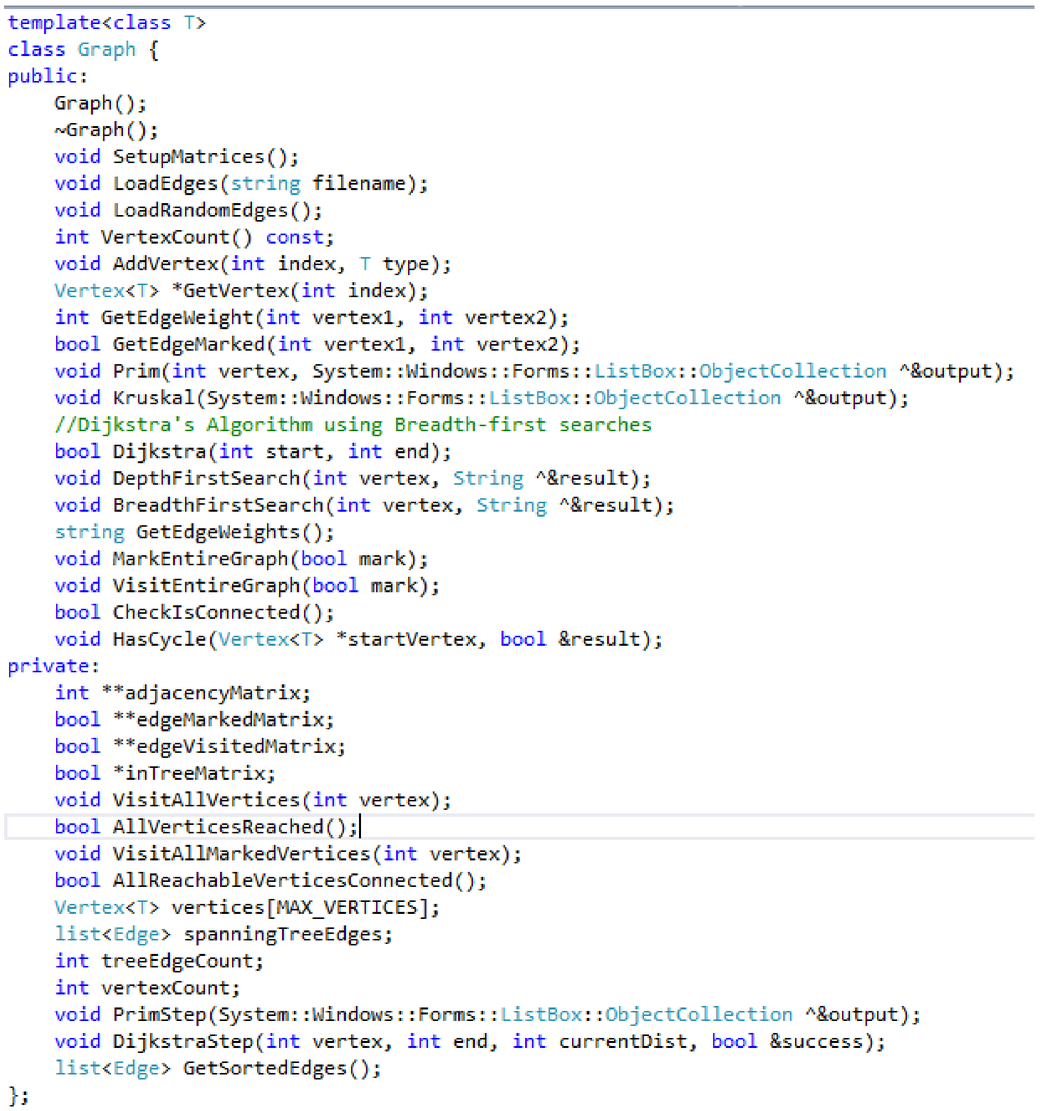
A graph is a data structure that stores data points as vertices, usually arranged arbitrarily. The vertices are connected by edges. These edges typically have weights, which represent the cost of traveling from one vertex to another, and also directions, indicating the path that the program can traverse in the graph. If a graph is undirected, each edge can be traversed in either direction by the program. Some algorithms that can be used on graphs are Prim's, Kruskal's, Dijkstra's, as well as breadth-first and depth-first searches. Breadh-first and depth-first searches, along with Dijkstra's algorithm were not too difficult to program, but Prim's and Kruskal's proved to be challenging.

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 a spanning tree or a path 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 are six text documents that contain several lines with 3 numbers on each line. Together, these files represent 3 unique graphs: a small, undirected graph, an acyclic graph and a cyclic graph. Three of the text files contain data on each vertex (index, position x and y), and the other three text files contain data on the graph's edges (starting vertex, end vertex, weight). After the program starts, the user can specify one of the three graphs to load into the program, using one of the three buttons. The user may also use the Random Graph button to generate a randomly arranged graph, with 30 vertices and 60 edges. The program then inserts the edges and vertices into the graph by calling its functions. The program then draws all of the graph nodes, scattered in arbitrary locations, and the connecting edges along with their weights. After the graph is loaded, the user may execute Prim's, Kruskal's, or Dijkstra's algorithm by using the text boxes and buttons provided. The user may also perform breadth-first and depth-first searches by using the controls near the bottom of the window.

Graph

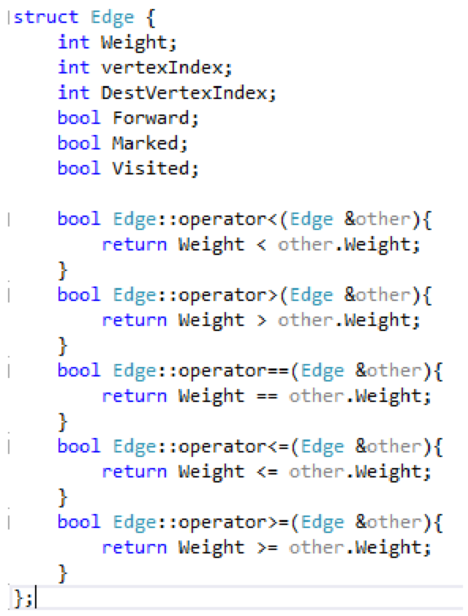
The program contains a Graph class. The program may execute Prim’s, Kruskal’s, and Dijkstra’s algorithms, detect whether or not the graph contains a cycle, perform a breadh-first or a depth-first search, or check whether all the vertices in the graph are interconnected. The Graph class can store vertices, and several matrices to represent edges in the graph, and whether or not those edges have been visited or marked by certain algorithms. The Graph is a template class, allowing the program to specify the template type of the vertices it will contain.



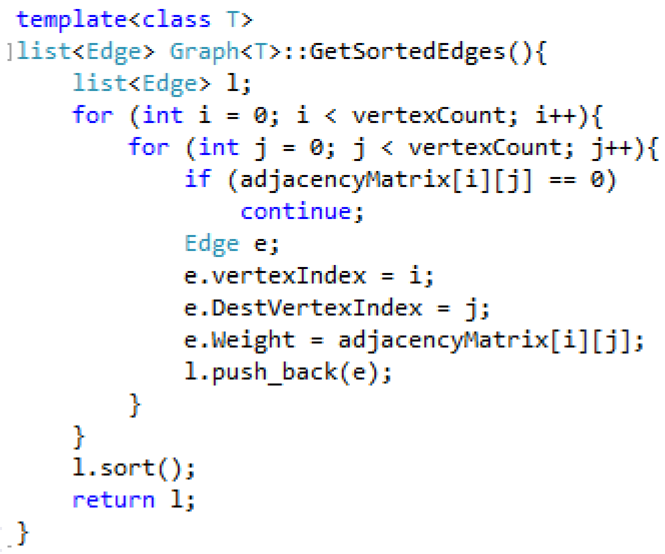
```
template<class T>
class Graph {
public:
    Graph();
    ~Graph();
    void SetupMatrices();
    void LoadEdges(string filename);
    void LoadRandomEdges();
    int VertexCount() const;
    void AddVertex(int index, T type);
    Vertex<T> *GetVertex(int index);
    int GetEdgeWeight(int vertex1, int vertex2);
    bool GetEdgeMarked(int vertex1, int vertex2);
    void Prim(int vertex, System::Windows::Forms::ListBox::ObjectCollection ^&output);
    void Kruskal(System::Windows::Forms::ListBox::ObjectCollection ^&output);
    //Dijkstra's Algorithm using Breadth-first searches
    bool Dijkstra(int start, int end);
    void DepthFirstSearch(int vertex, String ^&result);
    void BreadthFirstSearch(int vertex, String ^&result);
    string GetEdgeWeights();
    void MarkEntireGraph(bool mark);
    void VisitEntireGraph(bool mark);
    bool CheckIsConnected();
    void HasCycle(Vertex<T> *startVertex, bool &result);
private:
    int **adjacencyMatrix;
    bool **edgeMarkedMatrix;
    bool **edgeVisitedMatrix;
    bool *inTreeMatrix;
    void VisitAllVertices(int vertex);
    bool AllVerticesReached();
    void VisitAllMarkedVertices(int vertex);
    bool AllReachableVerticesConnected();
    Vertex<T> vertices[MAX_VERTICES];
    list<Edge> spanningTreeEdges;
    int treeEdgeCount;
    int vertexCount;
    void PrimStep(System::Windows::Forms::ListBox::ObjectCollection ^&output);
    void DijkstraStep(int vertex, int end, int currentDist, bool &success);
    list<Edge> GetSortedEdges();
};
```

Edges

Because edges in the graph are weighted, the Graph will sometimes need to visit edges on a vertex in order from lightest to heaviest. This happens in Prim's and Kruskal's algorithms, in which cases, the Graph needs to convert its adjacency matrix into Edge structures, then sort them in order of their weights. This is done in the Graph's GetSortedEdges function. The complexity of converting the matrices to edges in every case is O(n^2), with n being the number of vertices, therefore n^2 being the total number of edges in the graph. Even if all the vertices are not interconnected to each other vertex, the program still needs to iterate and check each space in the matrix, since zeros in the matrix represent two vertices that are not connected. The sorting algorithm for the list of edges is O(n log n) complexity for best and average case, and O(n^2) for worst case.



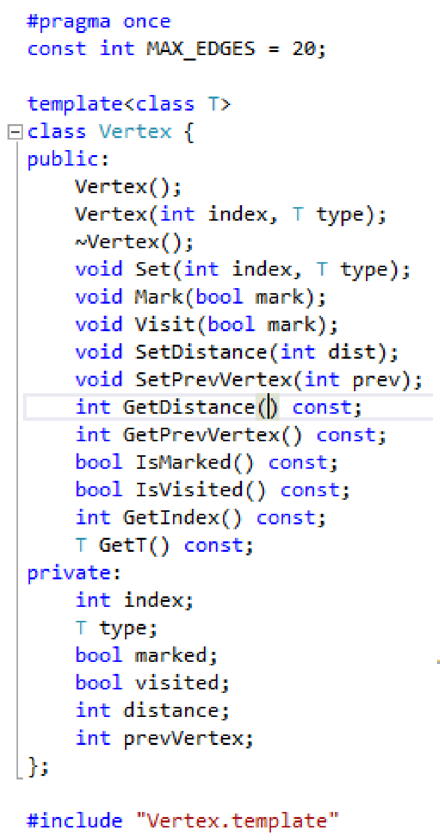
```
struct Edge {
    int Weight;
    int vertexIndex;
    int DestVertexIndex;
    bool Forward;
    bool Marked;
    bool Visited;

    bool Edge::operator<(Edge &other){
        return Weight < other.Weight;
    }
    bool Edge::operator>(Edge &other){
        return Weight > other.Weight;
    }
    bool Edge::operator==(Edge &other){
        return Weight == other.Weight;
    }
    bool Edge::operator<=(Edge &other){
        return Weight <= other.Weight;
    }
    bool Edge::operator>=(Edge &other){
        return Weight >= other.Weight;
    }
};
```



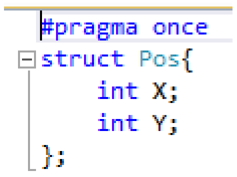
```
template<class T>
list<Edge> Graph<T>::GetSortedEdges(){
    list<Edge> l;
    for (int i = 0; i < vertexCount; i++){
        for (int j = 0; j < vertexCount; j++){
            if (adjacencyMatrix[i][j] == 0)
                continue;
            Edge e;
            e.vertexIndex = i;
            e.DestVertexIndex = j;
            e.Weight = adjacencyMatrix[i][j];
            l.push_back(e);
        }
    }
    l.sort();
    return l;
}
```

Vertices

The program contains a Vertex class to represent each vertex. Each vertex can be marked or visited, which is used when executing certain algorithms, like Prim's or Kruskal's. Each vertex can also be given a specified distance and an index to another vertex, which are used in Dijkstra's algorithm to measure traversal distance from one vertex to another. The Vertex class is generic, allowing it to store an arbitrary data type. In the program, the "T" type is used to store Pos structures, which keep track of a 2D position in space. This is used to determine where on the panel each vertex is drawn.



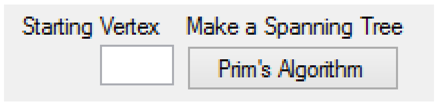
```
#pragma once
const int MAX_EDGES = 20;

template<class T>
class Vertex {
public:
    Vertex();
    Vertex(int index, T type);
    ~Vertex();
    void Set(int index, T type);
    void Mark(bool mark);
    void Visit(bool mark);
    void SetDistance(int dist);
    void SetPrevVertex(int prev);
    int GetDistance() const;
    int GetPrevVertex() const;
    bool IsMarked() const;
    bool IsVisited() const;
    int GetIndex() const;
    T GetT() const;
private:
    int index;
    T type;
    bool marked;
    bool visited;
    int distance;
    int prevVertex;
};

#include "Vertex.template"
```



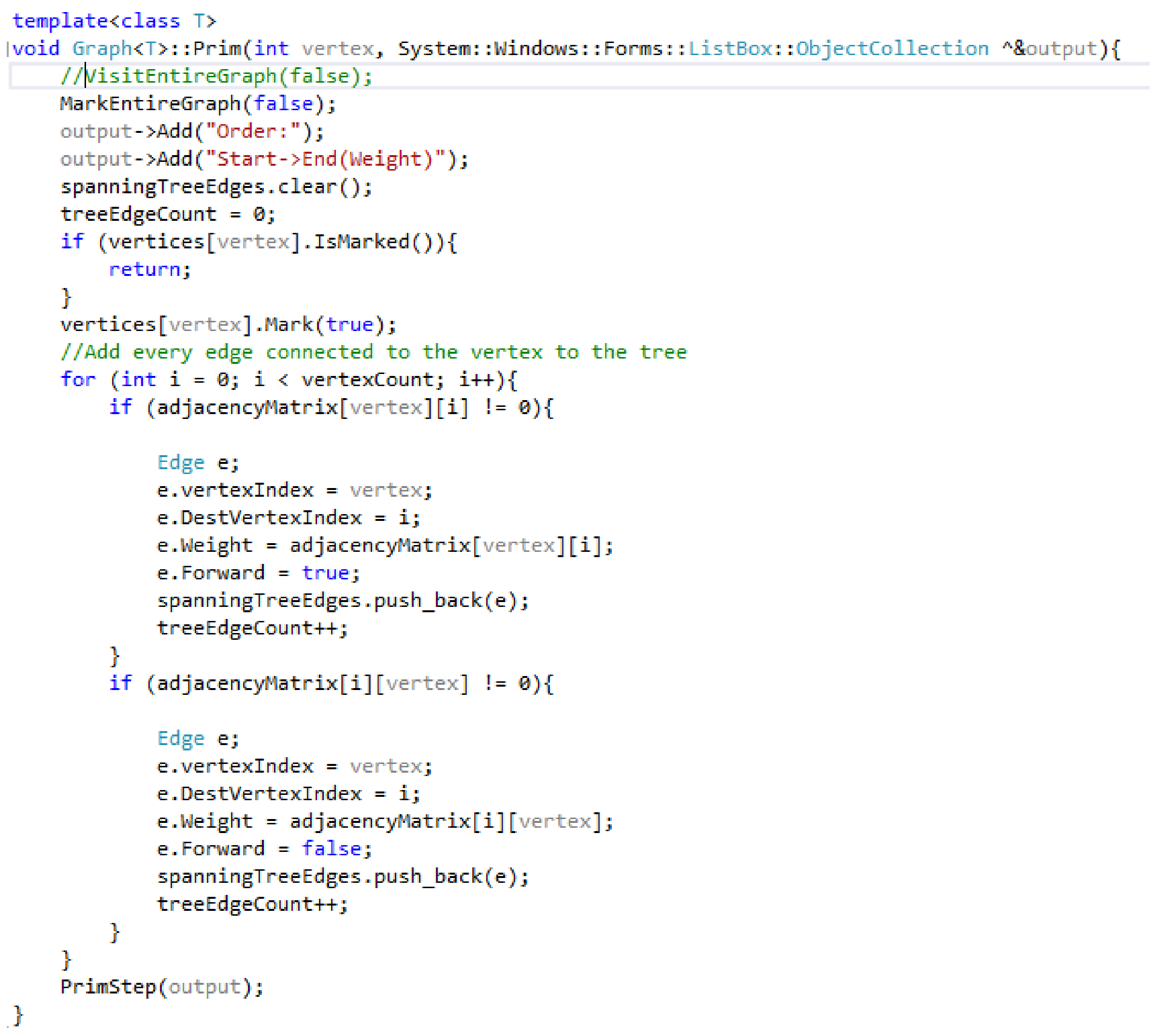
```
#pragma once
struct Pos{
    int X;
    int Y;
};
```

Prim’s Algorithm

Prim’s Algorithm is used to construct a minimum spanning tree on a graph. This is done by having the user select a vertex in the graph by specifying the vertex’s index, then pressing the button. The algorithm will not run if the user leaves Starting Vertex empty or if they enter an index that is out of range.



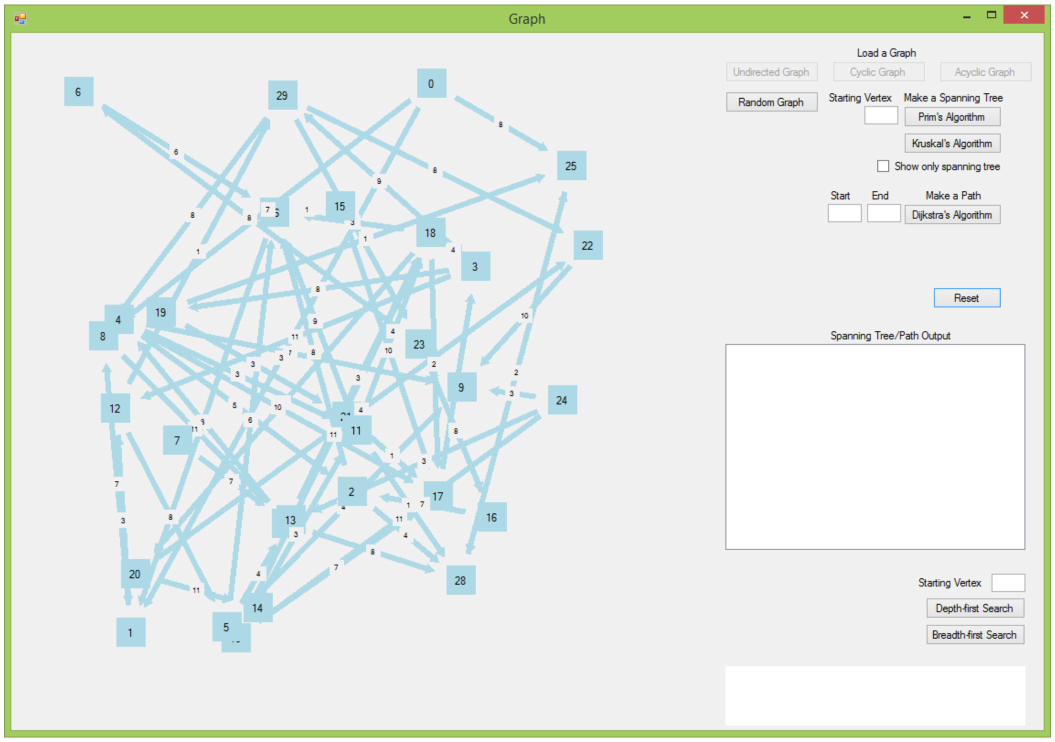
Before computing the algorithm, the program must first reset the Marked values for all edges and vertices, because they need to be marked when the algorithm runs. The complexity is O(n + n^2), (n for each vertex and n^2 for all edges), and is about the same for all cases, since it the program must iterate through all vertices and edges each time. The program then converts all edges connected to the starting vertex (regardless of direction) and converts them to Edge structures. The program sorts the edges from smallest to heaviest, chooses the edge with the smallest weight, and adds the vertex connected to the smallest edge into the tree. The process repeats until all vertices in the tree are connected. Iterating through each vertex is O(n), and iterating through each possible edge in the tree is O(n) for best case, O(n^2) for worst-case, and O((n/2)^2) for average-case. The complexity for checking each edge increases as the tree, and therefore the number of vertices increases. Each smallest edge and corresponding vertex is marked as they are added to the tree, so the graph knows which data points to highlight in dark blue.

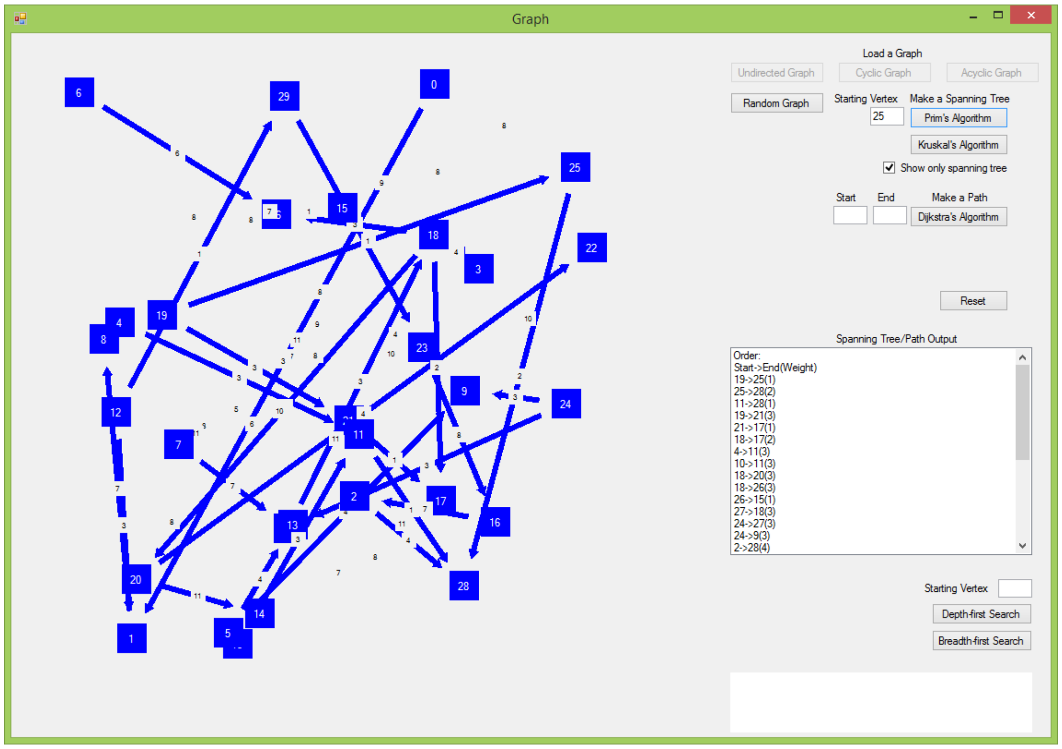


```
template<class T>
void Graph<T>::Prim(int vertex, System::Windows::Forms::ListBox::ObjectCollection ^&output){
    //VisitEntireGraph(false);
    MarkEntireGraph(false);
    output->Add("Order:");
    output->Add("Start->End(Weight)");
    spanningTreeEdges.clear();
    treeEdgeCount = 0;
    if (vertices[vertex].IsMarked()){
        return;
    }
    vertices[vertex].Mark(true);
    //Add every edge connected to the vertex to the tree
    for (int i = 0; i < vertexCount; i++){
        if (adjacencyMatrix[vertex][i] != 0){

            Edge e;
            e.vertexIndex = vertex;
            e.DestVertexIndex = i;
            e.Weight = adjacencyMatrix[vertex][i];
            e.Forward = true;
            spanningTreeEdges.push_back(e);
            treeEdgeCount++;
        }
        if (adjacencyMatrix[i][vertex] != 0){

            Edge e;
            e.vertexIndex = vertex;
            e.DestVertexIndex = i;
            e.Weight = adjacencyMatrix[i][vertex];
            e.Forward = false;
            spanningTreeEdges.push_back(e);
            treeEdgeCount++;
        }
    }
    PrimStep(output);
}
```

```
template<class T>
void Graph<T>::PrimStep(System::Windows::Forms::ListBox::ObjectCollection ^&output){
        list<Edge>::iterator it;
        Edge *smallestEdge;
        Vertex<T> *v = nullptr;
        int min = 99999;
        for (it = spanningTreeEdges.begin(); it != spanningTreeEdges.end(); it++){
                v = &vertices[(*it).DestVertexIndex];
                if ((*it).Weight < min && !v->IsMarked()){
                        min = (*it).Weight;
                        smallestEdge = &(*it);
                }
        }
        if (min != 99999){
                v = &vertices[smallestEdge->DestVertexIndex];
                v->Mark(true);
                //Does nothing
                smallestEdge->Marked = true;
                if (smallestEdge->Forward)
                        edgeMarkedMatrix[smallestEdge->vertexIndex][smallestEdge->DestVertexIndex] = true;
                else
                        edgeMarkedMatrix[smallestEdge->DestVertexIndex][smallestEdge->vertexIndex] = true;
        //Write to the output window
        if (smallestEdge->Forward)
            output->Add(smallestEdge->vertexIndex.ToString() + "->" + smallestEdge-
>DestVertexIndex.ToString() + "(" + smallestEdge->Weight.ToString() + ")");
        else
            output->Add(smallestEdge->DestVertexIndex.ToString() + "->" + smallestEdge-
>vertexIndex.ToString() + "(" + smallestEdge->Weight.ToString() + ")");

        //Add the edges connected to the vertex to the spanning tree
        for (int i = 0; i < vertexCount; i++){
            if (adjacencyMatrix[v->GetIndex()][i] != 0){
                Edge e;
                e.vertexIndex = v->GetIndex();
                e.DestVertexIndex = i;
                e.Weight = adjacencyMatrix[v->GetIndex()][i];
                e.Forward = true;
                spanningTreeEdges.push_back(e);
                treeEdgeCount++;
            }
            if (adjacencyMatrix[i][v->GetIndex()] != 0){
                Edge e;
                e.vertexIndex = v->GetIndex();
                e.DestVertexIndex = i;
                e.Weight = adjacencyMatrix[i][v->GetIndex()];
                e.Forward = false;
                spanningTreeEdges.push_back(e);
                treeEdgeCount++;
            }
        }
        PrimStep(output);
    }
    else{
        return;
    }
}
```

Sample Graph and Spanning tree using Prim’s algorithm, starting from vertex 25





Starting Vertex: 25

[✓] Show only spanning tree

Spanning Tree/Path Output

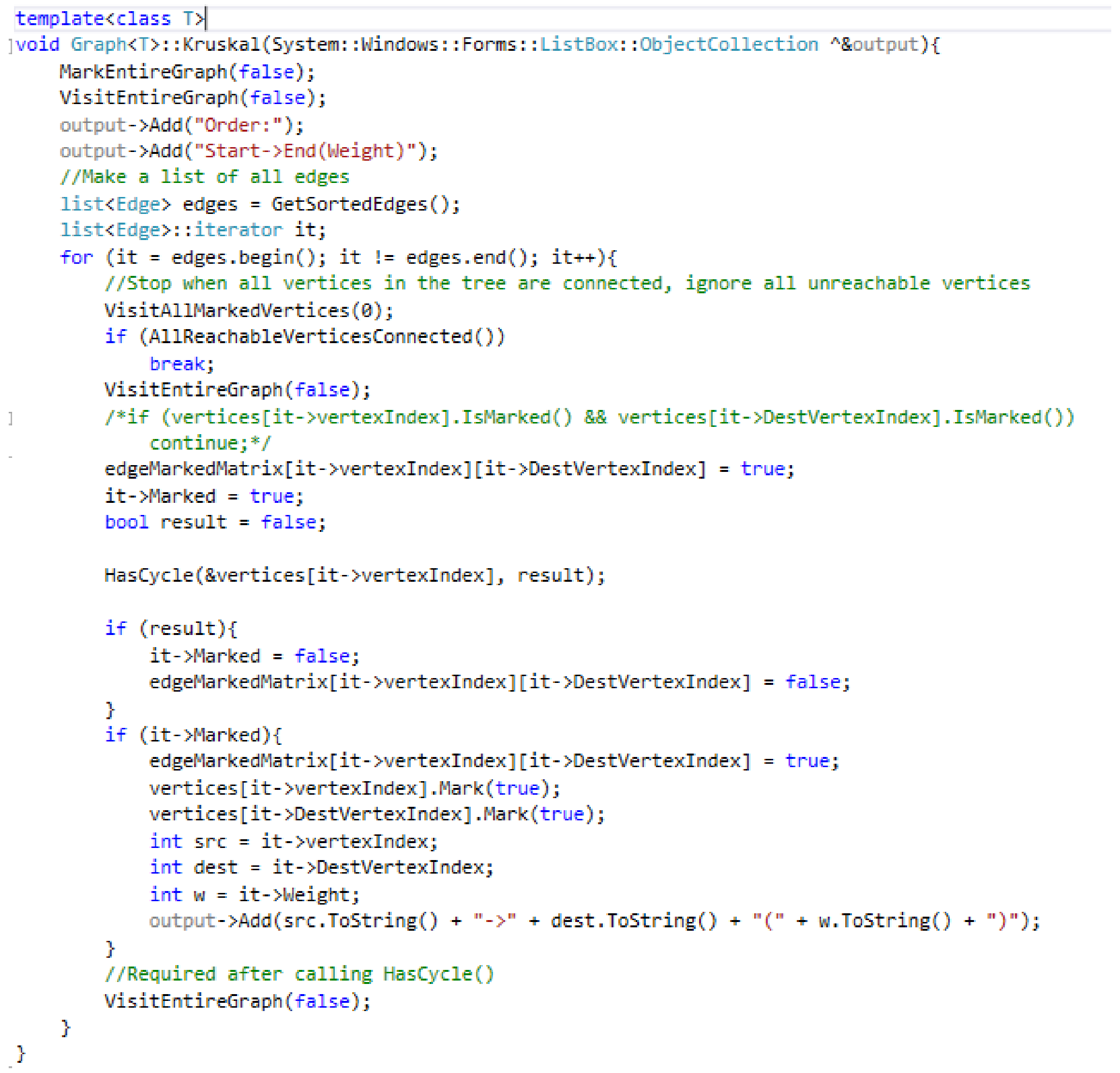
```
Order:
Start->End(Weight)
19->25(1)
25->28(2)
11->28(1)
19->21(3)
21->17(1)
18->17(2)
4->11(3)
10->11(3)
18->20(3)
18->26(3)
26->15(1)
27->18(3)
24->27(3)
24->9(3)
2->28(4)
```

Drawing edges and vertices

```
private: System::Void panel1_Paint(System::Object^  sender, System::Windows::Forms::PaintEventArgs^  e) {
    if (!graphLoaded)
        return;
    //Display edges
    for (int i = 0; i < graph->VertexCount(); i++){
        for (int j = 0; j < graph->VertexCount(); j++){
            int w = graph->GetEdgeWeight(i, j);
            if (w == 0)
                continue;
            Vertex<Pos> *v1 = graph->GetVertex(i);
            Vertex<Pos> *v2 = graph->GetVertex(j);
            int x1_ = v1->GetT().X;
            int y1_ = v1->GetT().Y;
            int x2_ = v2->GetT().X;
            int y2_ = v2->GetT().Y;

            float direction = Math::Atan2(y2_ - y1_, x2_ - x1_);

            int x1 = x1_ + (ARROW_OFFSET * Math::Cos(direction));
            int y1 = y1_ + (ARROW_OFFSET * Math::Sin(direction));
            int x2 = x2_ - (ARROW_OFFSET * Math::Cos(direction));
            int y2 = y2_ - (ARROW_OFFSET * Math::Sin(direction));

            if (!graph->GetEdgeMarked(i, j)){
                if (!checkBox1->Checked){
                    g->DrawLine(edgePen, x1 + 16, y1 + 16, x2 + 16, y2 + 16);
                }
            }
            else{
                g->DrawLine(markedEdgePen, x1 + 16, y1 + 16, x2 + 16, y2 + 16);
            }
        }
    }
}
```

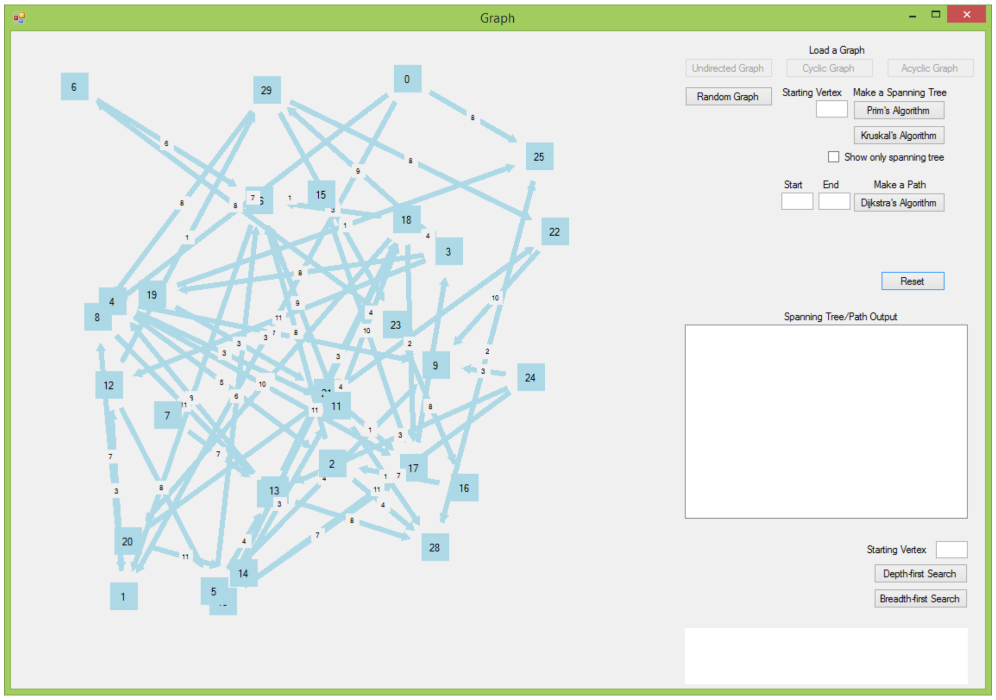
Kruskal's Algorithm

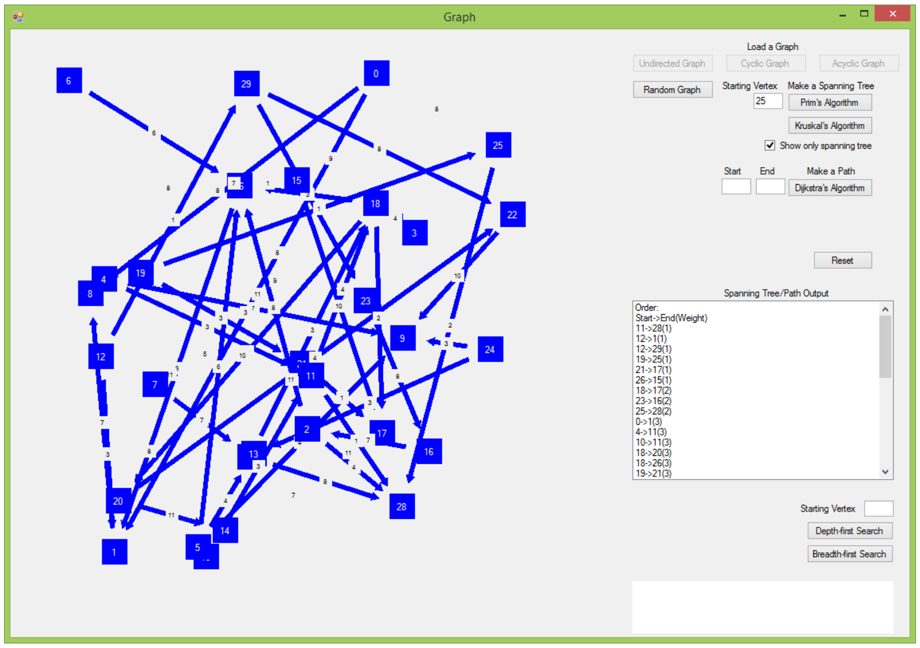
Kruskal's Algorithm works by searching through the edges in the graph from lightest to heaviest, and adds it to a spanning tree if it does not create a cycle in the graph. This continues until each vertex in the tree is in a single spanning tree. The program starts by getting a sorted list of all vertices in the graph with GetSortedEdges(). It then iterates through each vertex and marks it, along with its corresponding vertices, as long as it does not create a cycle in the graph. This continues until all vertices are in a single tree. The program uses the HasCycle function to check the edges for cycles. The complexity for HasCycle is O(n^2) for worst-case scenario, if it iterates through every edge on every vertex in the graph, O(n) for average-case scenario, if it iterates through all edges in a single branch, or O(1) for best-case scenario, if the edges immediately after the selected vertex creates a cycle



```
template<class T>
void Graph<T>::Kruskal(System::Windows::Forms::ListBox::ObjectCollection ^&output){
    MarkEntireGraph(false);
    VisitEntireGraph(false);
    output->Add("Order:");
    output->Add("Start->End(Weight)");
    //Make a list of all edges
    list<Edge> edges = GetSortedEdges();
    list<Edge>::iterator it;
    for (it = edges.begin(); it != edges.end(); it++){
        //Stop when all vertices in the tree are connected, ignore all unreachable vertices
        VisitAllMarkedVertices(0);
        if (AllReachableVerticesConnected())
            break;
        VisitEntireGraph(false);
        /*if (vertices[it->vertexIndex].IsMarked() && vertices[it->DestVertexIndex].IsMarked())
            continue;*/
        edgeMarkedMatrix[it->vertexIndex][it->DestVertexIndex] = true;
        it->Marked = true;
        bool result = false;

        HasCycle(&vertices[it->vertexIndex], result);

        if (result){
            it->Marked = false;
            edgeMarkedMatrix[it->vertexIndex][it->DestVertexIndex] = false;
        }
        if (it->Marked){
            edgeMarkedMatrix[it->vertexIndex][it->DestVertexIndex] = true;
            vertices[it->vertexIndex].Mark(true);
            vertices[it->DestVertexIndex].Mark(true);
            int src = it->vertexIndex;
            int dest = it->DestVertexIndex;
            int w = it->Weight;
            output->Add(src.ToString() + "->" + dest.ToString() + "(" + w.ToString() + ")");
        }
        //Required after calling HasCycle()
        VisitEntireGraph(false);
    }
}
```

```
template<class T>
void Graph<T>::HasCycle(Vertex<T> *vertex, bool &result) {
    if (vertex->IsVisited()){
        result = true;
        return;
    }
    vertex->Visit(true);
    for (int i = 0; i < vertexCount; i++){
        //Direction matters
        if (adjacencyMatrix[vertex->GetIndex()][i] != 0){
            if (!edgeVisitedMatrix[vertex->GetIndex()][i] && edgeMarkedMatrix[vertex->GetIndex()][i]){
                edgeVisitedMatrix[vertex->GetIndex()][i] = true;
                HasCycle(&vertices[i], result);
            }
        }
    }
}
```





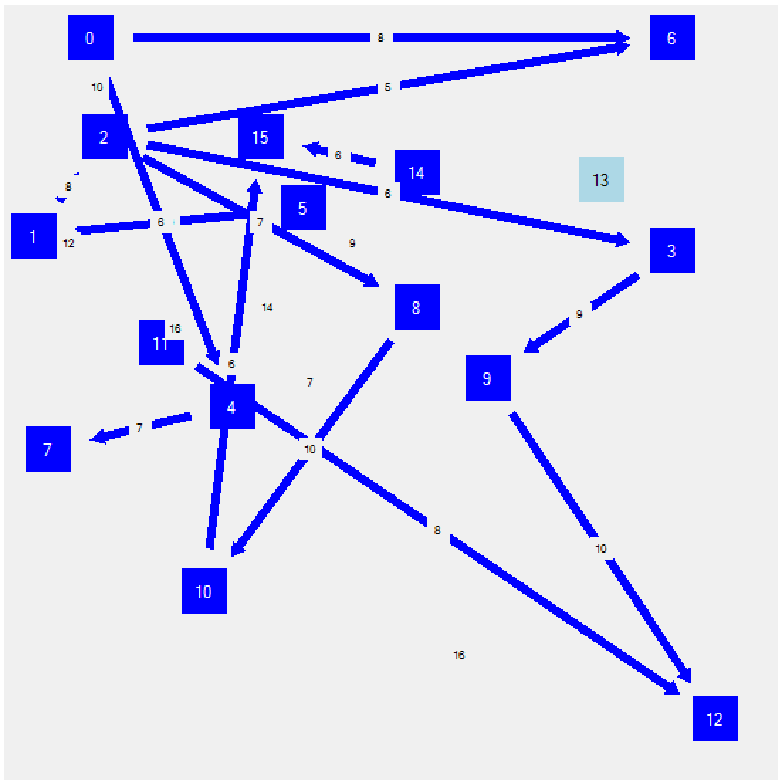
Starting Vertex 25

☑ Show only spanning tree

Spanning Tree/Path Output

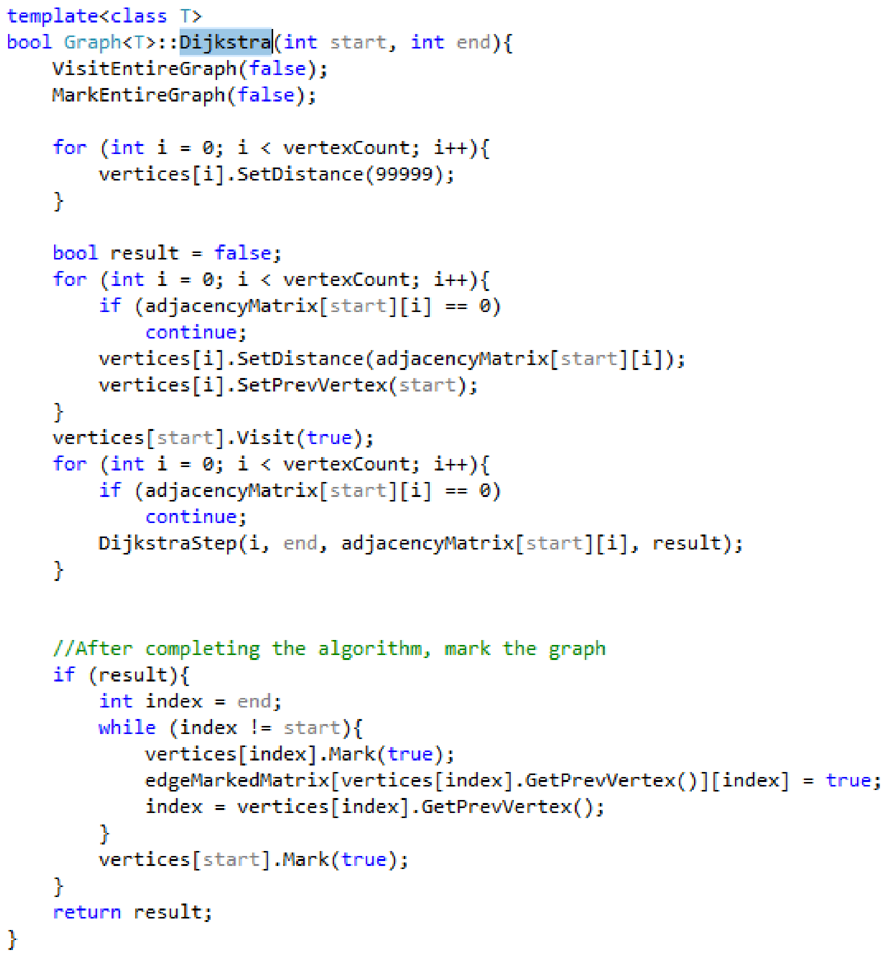
```
Order:
Start->End(Weight)
11->28(1)
12->1(1)
12->29(1)
19->25(1)
21->17(1)
26->15(1)
18->17(2)
23->16(2)
25->28(2)
0->1(3)
4->11(3)
10->11(3)
18->20(3)
18->26(3)
19->21(3)
```

The algorithm does not connect lone vertices



Dijkstra’s Algorithm

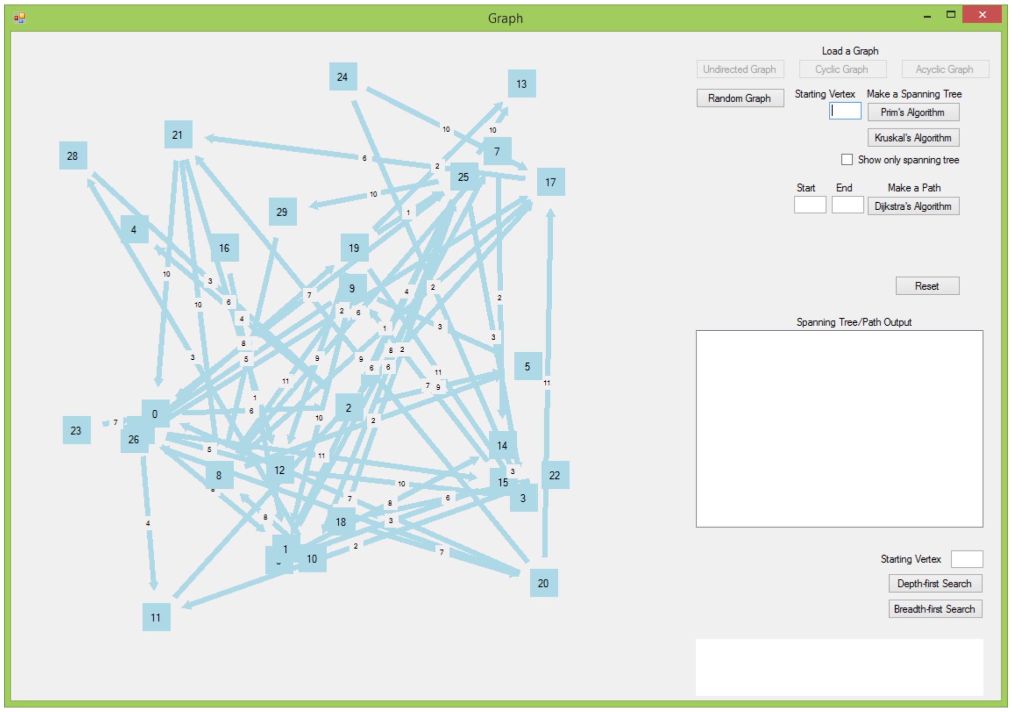
Dijkstra’s Algorithm is an algorithm that takes forms the shortest possible path between two vertices on a weighted graph. This is done by searching every vertex connected to a starting vertex, and marking the distance from that vertex to the starting vertex. This is done repeatedly until the end vertex has been reached. The worst-case complexity of searching each edge connected to each vertex is O(n), if the vertex is connected to all other vertices, average case is O(log n), if the vertex is connected to some of the other vertices, and best case is O(1) if the vertex is connected to only 1 other vertex.

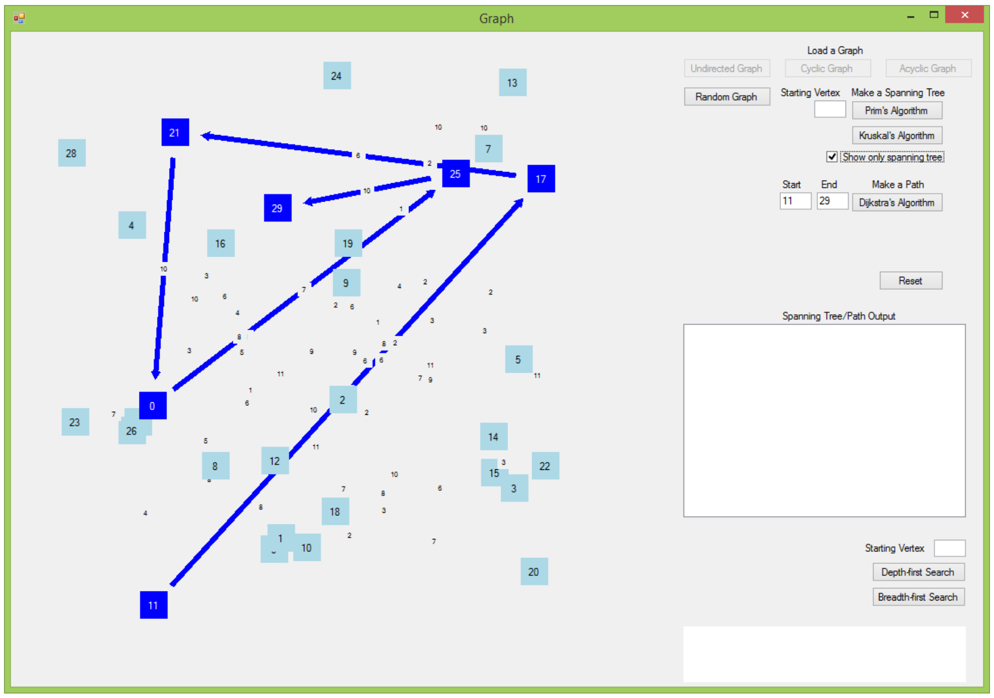


```cpp
template<class T>
bool Graph<T>::Dijkstra(int start, int end){
    VisitEntireGraph(false);
    MarkEntireGraph(false);

    for (int i = 0; i < vertexCount; i++){
        vertices[i].SetDistance(99999);
    }

    bool result = false;
    for (int i = 0; i < vertexCount; i++){
        if (adjacencyMatrix[start][i] == 0)
            continue;
        vertices[i].SetDistance(adjacencyMatrix[start][i]);
        vertices[i].SetPrevVertex(start);
    }
    vertices[start].Visit(true);
    for (int i = 0; i < vertexCount; i++){
        if (adjacencyMatrix[start][i] == 0)
            continue;
        DijkstraStep(i, end, adjacencyMatrix[start][i], result);
    }


    //After completing the algorithm, mark the graph
    if (result){
        int index = end;
        while (index != start){
            vertices[index].Mark(true);
            edgeMarkedMatrix[vertices[index].GetPrevVertex()][index] = true;
            index = vertices[index].GetPrevVertex();
        }
        vertices[start].Mark(true);
    }
    return result;
}
```

```
//Do not need to sort edges by length
template<class T>
void Graph<T>::DijkstraStep(int vertex, int end, int currentDist, bool &success){
    //Safety
    if (vertices[vertex].IsVisited())
        return;

    if (vertex == end){
        success = true;
        return;
    }

    for (int i = 0; i < vertexCount; i++){
        if (adjacencyMatrix[vertex][i] == 0 || vertices[i].IsMarked())
            continue;
        if (vertices[i].GetDistance() > currentDist + adjacencyMatrix[vertex][i]){
            vertices[i].SetDistance(currentDist + adjacencyMatrix[vertex][i]);
            vertices[i].SetPrevVertex(vertex);
        }
    }
    vertices[vertex].Visit(true);
    for (int i = 0; i < vertexCount; i++){
        if (adjacencyMatrix[vertex][i] == 0)
            continue;
        DijkstraStep(i, end, currentDist + adjacencyMatrix[vertex][i], success);
    }
}
```

Randomly generated graph





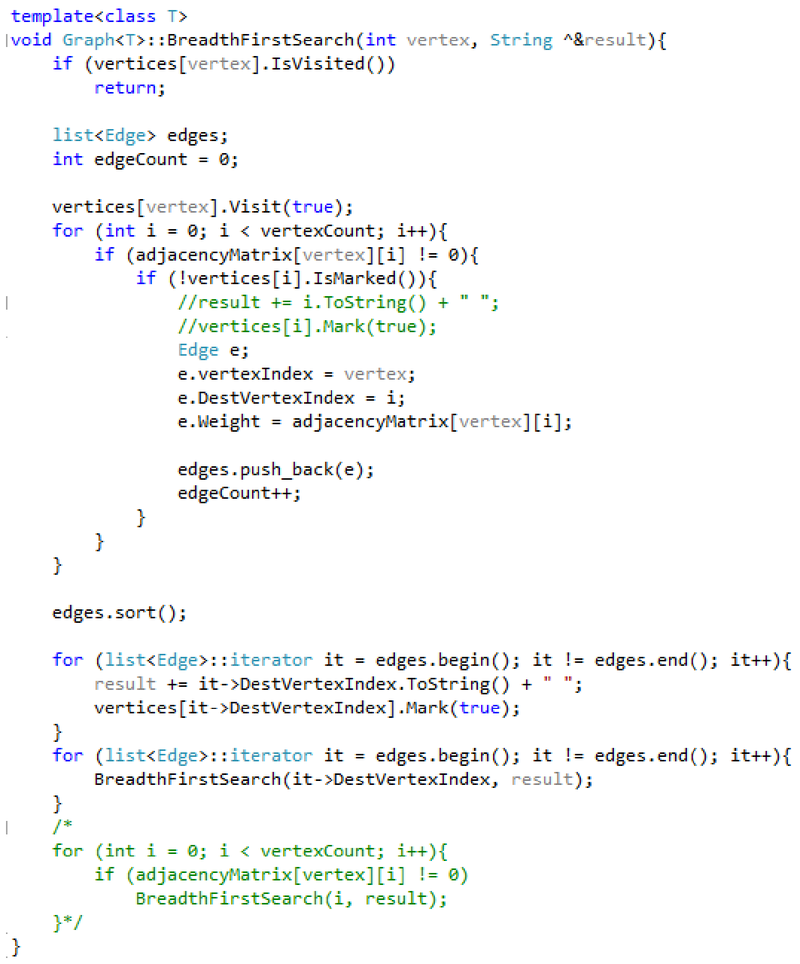
[✓] Show only spanning tree

Start: 11

End: 29

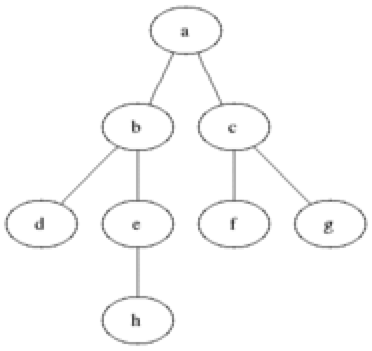
Breadth-first Search

A breadth-first search starts at a root vertex selected in the tree, explores the neighboring vertices first, then explores the neighbors of each of the neighbor vertices.

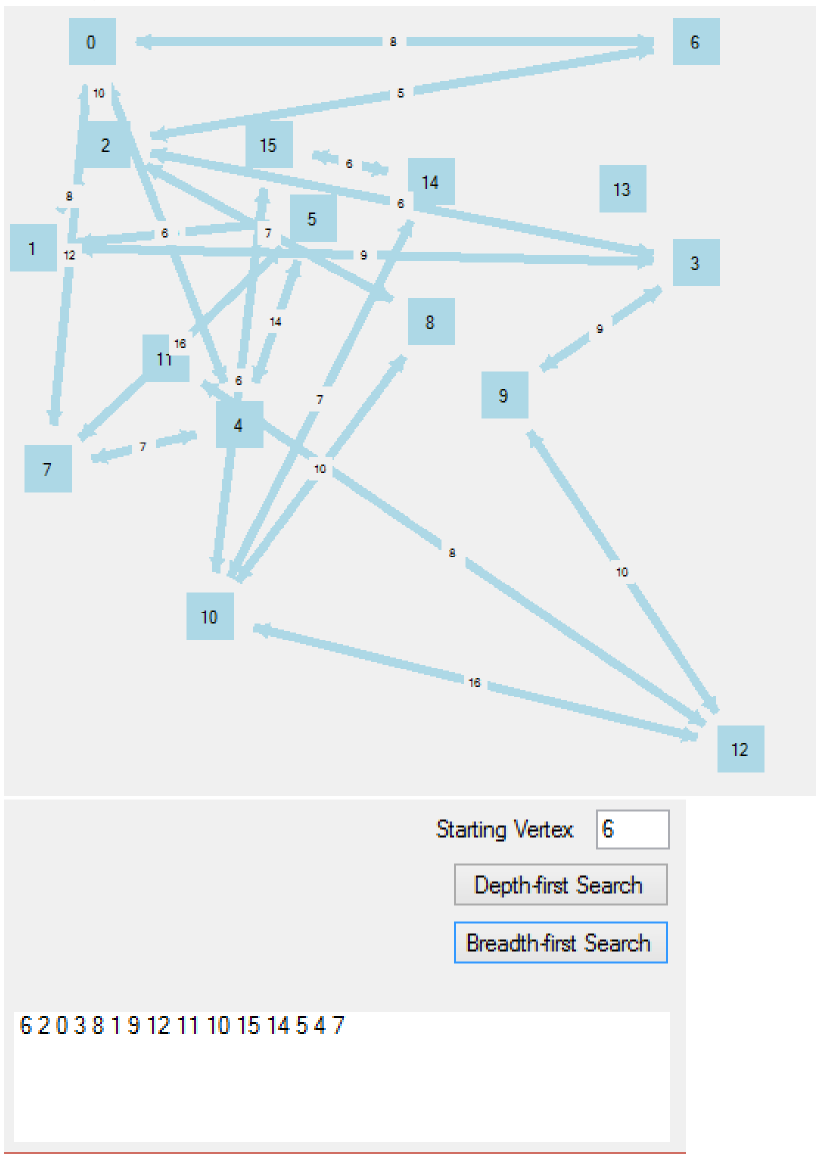


```
template<class T>
void Graph<T>::BreadthFirstSearch(int vertex, String ^&result){
    if (vertices[vertex].IsVisited())
        return;

    list<Edge> edges;
    int edgeCount = 0;

    vertices[vertex].Visit(true);
    for (int i = 0; i < vertexCount; i++){
        if (adjacencyMatrix[vertex][i] != 0){
            if (!vertices[i].IsMarked()){
                //result += i.ToString() + " ";
                //vertices[i].Mark(true);
                Edge e;
                e.vertexIndex = vertex;
                e.DestVertexIndex = i;
                e.Weight = adjacencyMatrix[vertex][i];

                edges.push_back(e);
                edgeCount++;
            }
        }
    }

    edges.sort();

    for (list<Edge>::iterator it = edges.begin(); it != edges.end(); it++){
        result += it->DestVertexIndex.ToString() + " ";
        vertices[it->DestVertexIndex].Mark(true);
    }
    for (list<Edge>::iterator it = edges.begin(); it != edges.end(); it++){
        BreadthFirstSearch(it->DestVertexIndex, result);
    }
    /*
    for (int i = 0; i < vertexCount; i++){
        if (adjacencyMatrix[vertex][i] != 0)
            BreadthFirstSearch(i, result);
    }*/
}
```

For instance, the vertices in this tree are explored in alphabetical order.



Small graph and breadth-first search order

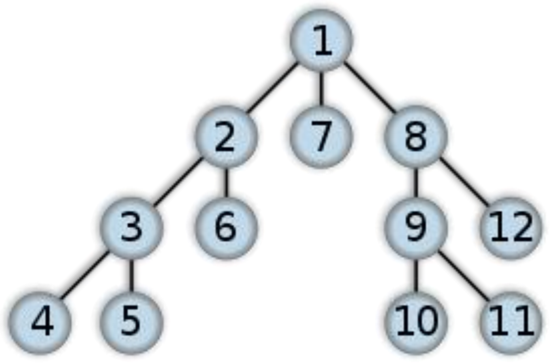


```
6 2 0 3 8 1 9 12 11 10 15 14 5 4 7
```

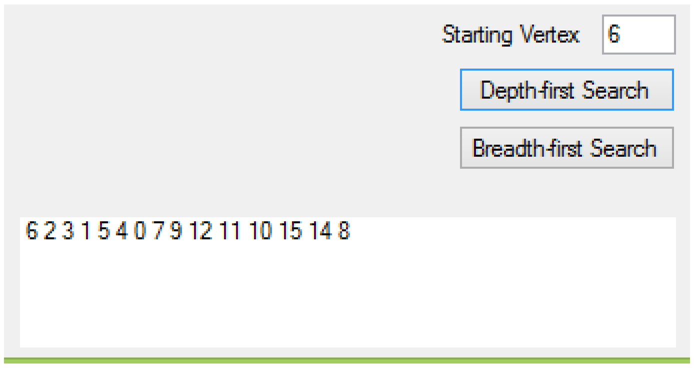
Depth-first Search

In contrast to a breadth-first search, a depth-first search explores as far as possible along one path of a neighboring vertex before exploring a path along another neighboring vertex.

```
//Should go in order of lightest to heaviest edge
template<class T>
void Graph<T>::DepthFirstSearch(int vertex, String ^&result){
    if (vertices[vertex].IsVisited())
        return;

    list<Edge> edges;
    int edgeCount = 0;

    vertices[vertex].Visit(true);
    result += vertex.ToString() + " ";
    for (int i = 0; i < vertexCount; i++){
        if (adjacencyMatrix[vertex][i] != 0){
            Edge e;
            e.vertexIndex = vertex;
            e.DestVertexIndex = i;
            e.Weight = adjacencyMatrix[vertex][i];

            edges.push_back(e);
            edgeCount++;
        }
    }

    edges.sort();

    for (list<Edge>::iterator it = edges.begin(); it != edges.end(); it++){
        DepthFirstSearch(it->DestVertexIndex, result);
    }
}
```



Depth-first search order from the same graph



Starting Vertex 6

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

Drawing Edges and Vertices

The LoadGraph function in GraphForm.h first determines how the graph is constructed with the given filename (graphName), and then it iterates through each vertex. It creates a label and adds it to the drawing panel in the form. Each label displays the index of the vertex, and its position given by the vertex's Pos field. Pos is a simple structure that contains x and y coordinates.

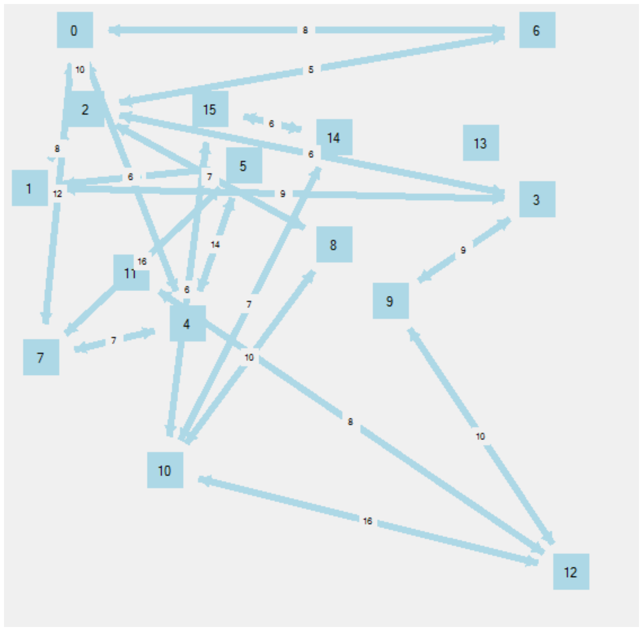
```
		void LoadGraph(string graphName){
			if (graph != nullptr){
				delete graph;
			}
			graph = new Graph<Pos>();
			LoadVertices(graph, graphName + " Graph Vertices.txt");
			graph->LoadEdges(graphName + " Graph Edges.txt");
			labels = gcnew cli::array<Label^>(graph->VertexCount());

			for (int i = 0; i < labelCount; i++){
				panel1->Controls->Remove(labels[i]);
			}
			labelCount = 0;

			//Display edge weights
			//O(n^2)
			for (int i = 0; i < graph->VertexCount(); i++){
				Label^ l = newLabel(i.ToString(), graph->GetVertex(i)-
>GetT().X, graph->GetVertex(i)->GetT().Y);
				labels[labelCount] = l;
				labelCount++;

				for (int j = 0; j < graph->VertexCount(); j++){
					int w = graph->GetEdgeWeight(i, j);
					if (w == 0)
						continue;
					Pos p1 = graph->GetVertex(i)->GetT();
					Pos p2 = graph->GetVertex(j)->GetT();
					int x1 = p1.X;
					int y1 = p1.Y;
					int x2 = p2.X;
					int y2 = p2.Y;
					Point center = Point(x1 + (x2 - x1) / 2, y1 + (y2 - y1)
/ 2);

					newLabel2(w.ToString(), center.X + 8, center.Y + 8);
				}
			}
			EnableGraphButtons(false);
			graphLoaded = true;
			panel1->Refresh();
		}
```

GraphForm.h contains an instance of the Graph class, and the panel1_Paint function draws each of the edges in the graph. This is done by iterating through each edge in the Graph's matrix and drawing it as an arrow. Light-blue lines represent edges not in a spanning tree or path, and dark-blue lines represent edges that are.

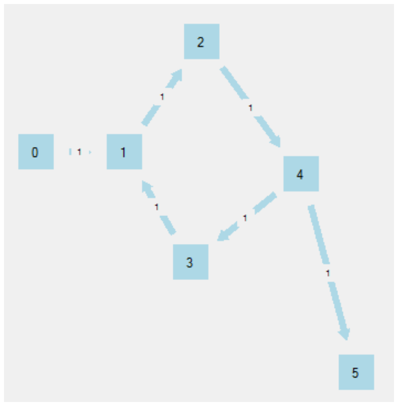
```
	private: System::Void panel1_Paint(System::Object^  sender, 
System::Windows::Forms::PaintEventArgs^  e) {
		if (!graphLoaded)
			return;
		//Display edges
		for (int i = 0; i < graph->VertexCount(); i++){
			for (int j = 0; j < graph->VertexCount(); j++){
				int w = graph->GetEdgeWeight(i, j);
				if (w == 0)
					continue;
				Vertex<Pos> *v1 = graph->GetVertex(i);
				Vertex<Pos> *v2 = graph->GetVertex(j);
				int x1_ = v1->GetT().X;
				int y1_ = v1->GetT().Y;
				int x2_ = v2->GetT().X;
				int y2_ = v2->GetT().Y;

				float direction = Math::Atan2(y2_ - y1_, x2_ - x1_);

				int x1 = x1_ + (ARROW_OFFSET * Math::Cos(direction));
				int y1 = y1_ + (ARROW_OFFSET * Math::Sin(direction));
				int x2 = x2_ - (ARROW_OFFSET * Math::Cos(direction));
				int y2 = y2_ - (ARROW_OFFSET * Math::Sin(direction));

				if (!graph->GetEdgeMarked(i, j)){
					if (!checkBox1->Checked){
						g->DrawLine(edgePen, x1 + 16, y1 + 16, x2 + 16, 
y2 + 16);
					}
				}
				else{
					g->DrawLine(markedEdgePen, x1 + 16, y1 + 16, x2 + 16, 
y2 + 16);
				}
			}
		}
	}
```

In addition to randomly-generated graphs, the program comes with 3 other graphs:

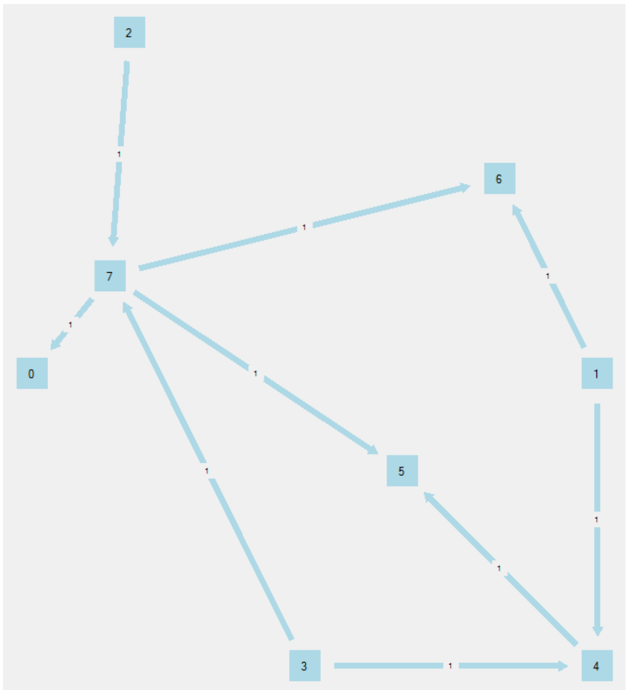
An undirected graph:



A cyclic directed graph:



And an acyclic directed graph:



Conclusion

Although I am quite familiar with all of these algorithms, Dijkstra and Prim turned out to be easier to program than Kruskal's. Because the stopping condition of the algorithm was checking to see if all vertices were connected in a single tree, and not multiple trees, I had to check for interconnectedness after adding every edge to the graph, which added a lot of complexity in addition to searching for cycles. I probably could have reduced some complexity by storing edges in a sorted list rather than a matrix. That way, when searching for a neighbor vertex, the program could search each edge as members of the Vertex, rather than searching through an entire row or column in a matrix. The program will skip over neighbors that the vertex is not connected to, and it is unlikely that a single vertex will be connected to all other vertices. Aside from that, these algorithms were very interesting to program, especially Kruskal's and Prim's. It is interesting to see a visual representation of these algorithms in a graph, and see that for any given graph, they seemed to produce the same result.

Graph source files

Acyclic Graph Edges.txt

Vertex	edge	weight
1	4	1
1	6	1
2	7	1
3	4	1
3	7	1
4	5	1
7	0	1
7	5	1
7	6	1

Acyclic Graph Vertices.txt

T	-	-
0	20	400
1	600	400
2	120	50
3	300	700
4	600	700
5	400	500
6	500	200
7	100	300

Cyclic Graph Edges.txt

Vertex	edge	weight
0	1	1
1	2	1
2	4	1
3	1	1
4	3	1
4	5	1

Cyclic Graph Vertices.txt

T	-	-
0	10	400
1	90	400
2	160	300
3	150	500
4	250	420
5	300	600

Small Graph Edges.txt

```
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

Small Graph Vertices.txt

T:15	-	-
0	40	10
1	0	150
2	50	80
3	450	160
4	140	270
5	190	130
6	450	10
7	10	300
8	270	200
9	320	250
10	120	400
11	90	225
12	480	490
13	400	110
14	270	105
15	160	80

Source code

Vertex.h

```
#pragma once
const int MAX_EDGES = 20;

template<class T>
class Vertex {
public:
	Vertex();
	Vertex(int index, T type);
	~Vertex();
	void Set(int index, T type);
	void Mark(bool mark);
	void Visit(bool mark);
	void SetDistance(int dist);
	void SetPrevVertex(int prev);
	int GetDistance() const;
	int GetPrevVertex() const;
	bool IsMarked() const;
	bool IsVisited() const;
	int GetIndex() const;
	T GetT() const;
private:
	int index;
	T type;
	bool marked;
	bool visited;
	int distance;
	int prevVertex;
};

#include "Vertex.template"
```

Vertex.template

```
template<class T>
Vertex<T>::Vertex(){
	marked = false;
	visited = false;
	index = 0;
}

template<class T>
Vertex<T>::Vertex(int index, T type) {
	marked = false;
	visited = false;
	Vertex::index = index;
	Vertex::type = type;
}

template<class T>
Vertex<T>::~Vertex(){

}
```

```
template<class T>
void Vertex<T>::Set(int index, T type){
	marked = false;
	visited = false;
	Vertex::index = index;
	Vertex::type = type;
}

template<class T>
void Vertex<T>::Mark(bool mark){
	marked = mark;
}

template<class T>
void Vertex<T>::Visit(bool mark){
	visited = mark;
}

template<class T>
void Vertex<T>::SetDistance(int dist){
	distance = dist;
}

template<class T>
void Vertex<T>::SetPrevVertex(int prev){
	prevVertex = prev;
}

template<class T>
int Vertex<T>::GetDistance() const{
	return distance;
}

template<class T>
int Vertex<T>::GetPrevVertex() const{
	return prevVertex;
}

template<class T>
bool Vertex<T>::IsMarked() const{
	return marked;
}

template<class T>
bool Vertex<T>::IsVisited() const{
	return visited;
}

template<class T>
int Vertex<T>::GetIndex() const
{
	return index;
}

template<class T>
T Vertex<T>::GetT() const{
	return type;
}
```

```
//template<class T>
//int Vertex::EdgeCount() const{
//	return edgeCount;
//}

//template<class T>
//Edge *Vertex::GetEdge(int index){
//	return (edges + index);
//}
```

Graph.h

```
#pragma once
#include "Vertex.h"
#include <string>
#include <fstream>
#include <list>
using namespace std;
using namespace System;

const int MAX_VERTICES = 50;

struct Edge {
	int Weight;
	int vertexIndex;
	int DestVertexIndex;
	bool Forward;
	bool Marked;
	bool Visited;

	bool Edge::operator<(Edge &other){
		return Weight < other.Weight;
	}
	bool Edge::operator>(Edge &other){
		return Weight > other.Weight;
	}
	bool Edge::operator==(Edge &other){
		return Weight == other.Weight;
	}
	bool Edge::operator<=(Edge &other){
		return Weight <= other.Weight;
	}
	bool Edge::operator>=(Edge &other){
		return Weight >= other.Weight;
	}
};

template<class T>
class Graph {
public:
	Graph();
	~Graph();
	void SetupMatrices();
	void LoadEdges(string filename);
	void LoadRandomEdges();
	int VertexCount() const;
	void AddVertex(int index, T type);
	Vertex<T> *GetVertex(int index);
	int GetEdgeWeight(int vertex1, int vertex2);
	bool GetEdgeMarked(int vertex1, int vertex2);
	void Prim(int vertex, System::Windows::Forms::ListBox::ObjectCollection ^&output);
	void Kruskal(System::Windows::Forms::ListBox::ObjectCollection ^&output);
	//Dijkstra's Algorithm using Breadth-first searches
	bool Dijkstra(int start, int end);
	void DepthFirstSearch(int vertex, String ^&result);
	void BreadthFirstSearch(int vertex, String ^&result);
	string GetEdgeWeights();
	void MarkEntireGraph(bool mark);
	void VisitEntireGraph(bool mark);
	bool CheckIsConnected();
	void HasCycle(Vertex<T> *startVertex, bool &result);
private:
	int **adjacencyMatrix;
	bool **edgeMarkedMatrix;
	bool **edgeVisitedMatrix;
	bool *inTreeMatrix;
	void VisitAllVertices(int vertex);
	bool AllVerticesReached();
	void VisitAllMarkedVertices(int vertex);
	bool AllReachableVerticesConnected();
	Vertex<T> vertices[MAX_VERTICES];
	list<Edge> spanningTreeEdges;
	int treeEdgeCount;
	int vertexCount;
	void PrimStep(System::Windows::Forms::ListBox::ObjectCollection ^&output);
	void DijkstraStep(int vertex, int end, int currentDist, bool &success);
	list<Edge> GetSortedEdges();
};

#include "Graph.template"
```

Graph.template

```
template<class T>
Graph<T>::Graph(){
	vertexCount = 0;
	treeEdgeCount = 0;
}

template<class T>
Graph<T>::~Graph(){
	for (int i = 0; i < vertexCount; i++){
		delete[] adjacencyMatrix[i];
		delete[] edgeMarkedMatrix[i];
		delete[] edgeVisitedMatrix[i];
	}
	delete[] inTreeMatrix;
	delete[] adjacencyMatrix;
	delete[] edgeMarkedMatrix;
	delete[] edgeVisitedMatrix;
}

template<class T>
void Graph<T>::SetupMatrices(){
	//First index should be source vertex, second should be destination vertex
	adjacencyMatrix = new int*[vertexCount];
	edgeMarkedMatrix = new bool*[vertexCount];
	edgeVisitedMatrix = new bool*[vertexCount];
	for (int i = 0; i < vertexCount; i++){
		inTreeMatrix = new bool[vertexCount];
		adjacencyMatrix[i] = new int[vertexCount];
		edgeMarkedMatrix[i] = new bool[vertexCount];
		edgeVisitedMatrix[i] = new bool[vertexCount];
		for (int j = 0; j < vertexCount; j++){
			adjacencyMatrix[i][j] = 0;
			edgeMarkedMatrix[i][j] = false;
			edgeVisitedMatrix[i][j] = false;
		}
	}
	for (int i = 0; i < vertexCount; i++){
		inTreeMatrix[i] = true;
	}
}

template<class T>
void Graph<T>::LoadEdges(string filename){
	ifstream file;
	file.open(filename);
	if (file.fail()) {
		file.close();
		System::Windows::Forms::Application::Exit();
	}
	while (!file.eof()) {
		string n1, n2, w;
		file >> n1 >> n2 >> w;
		if (n1 != "" && n1 != "Vertex")
		{
			adjacencyMatrix[stoi(n1)][stoi(n2)] = stoi(w);
		}
	}
	CheckIsConnected();
	file.close();
}

template<class T>
void Graph<T>::LoadRandomEdges() {
	for (int i = 0; i < 60; i++){
		int v1 = -1, v2 = -1;
		do{
			v1 = rand() % vertexCount;
			v2 = rand() % vertexCount;
		} while (adjacencyMatrix[v1][v2] != 0);
		adjacencyMatrix[v1][v2] = 1 + rand() % 11;
	}
	for (int i = 0; i < vertexCount; i++){
		adjacencyMatrix[i][i] = 0;
	}
	//bool connected = false;
	//do{
	//	VisitAllVertices(false);
	//	int v1 = rand() % vertexCount;
	//	int v2 = rand() % vertexCount;
	//	adjacencyMatrix[v1][v2] = 1 + rand() % 11;	//1 - 10
	//	VisitAllVertices(0);
	//	connected = AllVerticesReached();
	//} while (!connected);
	//VisitAllVertices(false);
}

template<class T>
void Graph<T>::VisitAllVertices(int vertex){
	if (vertices[vertex].IsVisited())
		return;

	vertices[vertex].Visit(true);
	for (int i = 0; i < vertexCount; i++){
		if (adjacencyMatrix[vertex][i] != 0 || adjacencyMatrix[i][vertex] != 0)
			VisitAllVertices(i);
	}
}

template<class T>
bool Graph<T>::AllVerticesReached(){
	for (int i = 0; i < vertexCount; i++){
		if (!vertices[i].IsVisited()){
			return false;
		}
	}
	return true;
}

template<class T>
int Graph<T>::VertexCount() const{
	return vertexCount;
}

template<class T>
void Graph<T>::AddVertex(int index, T type){
	vertices[vertexCount].Set(index, type);
	vertexCount++;
}

template<class T>
Vertex<T> *Graph<T>::GetVertex(int index) {
	return &vertices[index];
}

template<class T>
int Graph<T>::GetEdgeWeight(int vertex1, int vertex2){
	return adjacencyMatrix[vertex1][vertex2];
}

template<class T>
bool Graph<T>::GetEdgeMarked(int vertex1, int vertex2){
	return edgeMarkedMatrix[vertex1][vertex2];
}

template<class T>
void Graph<T>::Prim(int vertex, System::Windows::Forms::ListBox::ObjectCollection
^&output){
	//VisitEntireGraph(false);
	MarkEntireGraph(false);
	output->Add("Order:");
	output->Add("Start->End(Weight)");
	spanningTreeEdges.clear();
	treeEdgeCount = 0;
	if (vertices[vertex].IsMarked()){
		return;
	}
	vertices[vertex].Mark(true);
	//Add every edge connected to the vertex to the tree
	for (int i = 0; i < vertexCount; i++){
		if (adjacencyMatrix[vertex][i] != 0){

			Edge e;
			e.vertexIndex = vertex;
			e.DestVertexIndex = i;
			e.Weight = adjacencyMatrix[vertex][i];
			e.Forward = true;
			spanningTreeEdges.push_back(e);
			treeEdgeCount++;
		}
		if (adjacencyMatrix[i][vertex] != 0){

			Edge e;
			e.vertexIndex = vertex;
			e.DestVertexIndex = i;
			e.Weight = adjacencyMatrix[i][vertex];
			e.Forward = false;
			spanningTreeEdges.push_back(e);
			treeEdgeCount++;
		}
	}
	PrimStep(output);
}

template<class T>
void Graph<T>::PrimStep(System::Windows::Forms::ListBox::ObjectCollection ^&output){
	list<Edge>::iterator it;
	Edge *smallestEdge;
	Vertex<T> *v = nullptr;
	int min = 99999;
	for (it = spanningTreeEdges.begin(); it != spanningTreeEdges.end(); it++){
		v = &vertices[(*it).DestVertexIndex];
		if ((*it).Weight < min && !v->IsMarked()){
			min = (*it).Weight;
			smallestEdge = &(*it);
		}
	}
	if (min != 99999){
		v = &vertices[smallestEdge->DestVertexIndex];
		v->Mark(true);
		//Does nothing
		smallestEdge->Marked = true;
		if (smallestEdge->Forward)
			edgeMarkedMatrix[smallestEdge->vertexIndex][smallestEdge-
>DestVertexIndex] = true;
		else
```

```
                        edgeMarkedMatrix[smallestEdge->DestVertexIndex][smallestEdge-
>vertexIndex] = true;

                //Write to the output window
                if (smallestEdge->Forward)
                        output->Add(smallestEdge->vertexIndex.ToString() + "->" +
smallestEdge->DestVertexIndex.ToString() + "(" + smallestEdge->Weight.ToString() + ")");
                else
                        output->Add(smallestEdge->DestVertexIndex.ToString() + "->" +
smallestEdge->vertexIndex.ToString() + "(" + smallestEdge->Weight.ToString() + ")");

                //Add the edges connected to the vertex to the spanning tree
                for (int i = 0; i < vertexCount; i++){
                        if (adjacencyMatrix[v->GetIndex()][i] != 0){
                                Edge e;
                                e.vertexIndex = v->GetIndex();
                                e.DestVertexIndex = i;
                                e.Weight = adjacencyMatrix[v->GetIndex()][i];
                                e.Forward = true;
                                spanningTreeEdges.push_back(e);
                                treeEdgeCount++;
                        }
                        if (adjacencyMatrix[i][v->GetIndex()] != 0){
                                Edge e;
                                e.vertexIndex = v->GetIndex();
                                e.DestVertexIndex = i;
                                e.Weight = adjacencyMatrix[i][v->GetIndex()];
                                e.Forward = false;
                                spanningTreeEdges.push_back(e);
                                treeEdgeCount++;
                        }
                }
                PrimStep(output);
        }
        else{
                return;
        }
}

template<class T>
list<Edge> Graph<T>::GetSortedEdges(){
        list<Edge> l;
        for (int i = 0; i < vertexCount; i++){
                for (int j = 0; j < vertexCount; j++){
                        if (adjacencyMatrix[i][j] == 0)
                                continue;
                        Edge e;
                        e.vertexIndex = i;
                        e.DestVertexIndex = j;
                        e.Weight = adjacencyMatrix[i][j];
                        l.push_back(e);
                }
        }
        l.sort();
        return l;
}

template<class T>
```

```
void Graph<T>::VisitAllMarkedVertices(int v){
	//Check if all vertices in the tree are connected
	if (vertices[v].IsVisited())
		return;

	vertices[v].Visit(true);
	for (int i = 0; i < vertexCount; i++){
		if ((edgeMarkedMatrix[v][i] || edgeMarkedMatrix[i][v]) && inTreeMatrix[i]){
			VisitAllMarkedVertices(i);
		}
	}
}

template<class T>
bool Graph<T>::AllReachableVerticesConnected(){
	for (int i = 0; i < vertexCount; i++){
		if (inTreeMatrix[i]){
			if (!vertices[i].IsVisited())
				return false;
		}
	}
	return true;
}

template<class T>
void Graph<T>::Kruskal(System::Windows::Forms::ListBox::ObjectCollection ^&output){
	MarkEntireGraph(false);
	VisitEntireGraph(false);
	output->Add("Order:");
	output->Add("Start->End(Weight)");
	//Make a list of all edges
	list<Edge> edges = GetSortedEdges();
	list<Edge>::iterator it;
	for (it = edges.begin(); it != edges.end(); it++){
		//Stop when all vertices in the tree are connected, ignore all unreachable
vertices
		VisitAllMarkedVertices(0);
		if (AllReachableVerticesConnected())
			break;
		VisitEntireGraph(false);
		/*if (vertices[it->vertexIndex].IsMarked() && vertices[it-
>DestVertexIndex].IsMarked())
			continue;*/
		edgeMarkedMatrix[it->vertexIndex][it->DestVertexIndex] = true;
		it->Marked = true;
		bool result = false;

		HasCycle(&vertices[it->vertexIndex], result);

		if (result){
			it->Marked = false;
			edgeMarkedMatrix[it->vertexIndex][it->DestVertexIndex] = false;
		}
		if (it->Marked){
			edgeMarkedMatrix[it->vertexIndex][it->DestVertexIndex] = true;
			vertices[it->vertexIndex].Mark(true);
			vertices[it->DestVertexIndex].Mark(true);
			int src = it->vertexIndex;
```

```
                        int dest = it->DestVertexIndex;
                        int w = it->Weight;
                        output->Add(src.ToString() + "->" + dest.ToString() + "(" +
w.ToString() + ")");
                }
                //Required after calling HasCycle()
                VisitEntireGraph(false);
        }
}

template<class T>
void Graph<T>::HasCycle(Vertex<T> *vertex, bool &result) {
        if (vertex->IsVisited()){
                result = true;
                return;
        }
        vertex->Visit(true);
        for (int i = 0; i < vertexCount; i++){
                //Direction matters
                if (adjacencyMatrix[vertex->GetIndex()][i] != 0){
                        if (!edgeVisitedMatrix[vertex->GetIndex()][i] &&
edgeMarkedMatrix[vertex->GetIndex()][i]){
                                edgeVisitedMatrix[vertex->GetIndex()][i] = true;
                                HasCycle(&vertices[i], result);
                        }
                }
        }
}

template<class T>
bool Graph<T>::Dijkstra(int start, int end){
        VisitEntireGraph(false);
        MarkEntireGraph(false);

        for (int i = 0; i < vertexCount; i++){
                vertices[i].SetDistance(99999);
        }

        bool result = false;
        for (int i = 0; i < vertexCount; i++){
                if (adjacencyMatrix[start][i] == 0)
                        continue;
                vertices[i].SetDistance(adjacencyMatrix[start][i]);
                vertices[i].SetPrevVertex(start);
        }
        vertices[start].Visit(true);
        for (int i = 0; i < vertexCount; i++){
                if (adjacencyMatrix[start][i] == 0)
                        continue;
                DijkstraStep(i, end, adjacencyMatrix[start][i], result);
        }


        //After completing the algorithm, mark the graph
        if (result){
                int index = end;
                while (index != start){
                        vertices[index].Mark(true);
                edgeMarkedMatrix[vertices[index].GetPrevVertex()][index] = true;
                index = vertices[index].GetPrevVertex();
            }
            vertices[start].Mark(true);
        }
        return result;
}

//Do not need to sort edges by length
template<class T>
void Graph<T>::DijkstraStep(int vertex, int end, int currentDist, bool &success){
        //Safety
        if (vertices[vertex].IsVisited())
            return;

        if (vertex == end){
            success = true;
            return;
        }

        for (int i = 0; i < vertexCount; i++){
            if (adjacencyMatrix[vertex][i] == 0 || vertices[i].IsMarked())
                continue;
            if (vertices[i].GetDistance() > currentDist + adjacencyMatrix[vertex][i]){
                vertices[i].SetDistance(currentDist + adjacencyMatrix[vertex][i]);
                vertices[i].SetPrevVertex(vertex);
            }
        }
        vertices[vertex].Visit(true);
        for (int i = 0; i < vertexCount; i++){
            if (adjacencyMatrix[vertex][i] == 0)
                continue;
            DijkstraStep(i, end, currentDist + adjacencyMatrix[vertex][i], success);
        }
}

//Should go in order of lightest to heaviest edge
template<class T>
void Graph<T>::DepthFirstSearch(int vertex, String ^&result){
        if (vertices[vertex].IsVisited())
            return;

        list<Edge> edges;
        int edgeCount = 0;

        vertices[vertex].Visit(true);
        result += vertex.ToString() + " ";
        for (int i = 0; i < vertexCount; i++){
            if (adjacencyMatrix[vertex][i] != 0){
                Edge e;
                e.vertexIndex = vertex;
                e.DestVertexIndex = i;
                e.Weight = adjacencyMatrix[vertex][i];

                edges.push_back(e);
                edgeCount++;
            }
        }
        edges.sort();

        for (list<Edge>::iterator it = edges.begin(); it != edges.end(); it++){
                DepthFirstSearch(it->DestVertexIndex, result);
        }
}

template<class T>
void Graph<T>::BreadthFirstSearch(int vertex, String ^&result){
        if (vertices[vertex].IsVisited())
                return;

        list<Edge> edges;
        int edgeCount = 0;

        vertices[vertex].Visit(true);
        for (int i = 0; i < vertexCount; i++){
                if (adjacencyMatrix[vertex][i] != 0){
                        if (!vertices[i].IsMarked()){
                                //result += i.ToString() + " ";
                                //vertices[i].Mark(true);
                                Edge e;
                                e.vertexIndex = vertex;
                                e.DestVertexIndex = i;
                                e.Weight = adjacencyMatrix[vertex][i];

                                edges.push_back(e);
                                edgeCount++;
                        }
                }
        }

        edges.sort();

        for (list<Edge>::iterator it = edges.begin(); it != edges.end(); it++){
                result += it->DestVertexIndex.ToString() + " ";
                vertices[it->DestVertexIndex].Mark(true);
        }
        for (list<Edge>::iterator it = edges.begin(); it != edges.end(); it++){
                BreadthFirstSearch(it->DestVertexIndex, result);
        }
        /*
        for (int i = 0; i < vertexCount; i++){
                if (adjacencyMatrix[vertex][i] != 0)
                        BreadthFirstSearch(i, result);
        }*/
}

template<class T>
string Graph<T>::GetEdgeWeights(){
        list<Edge> edges = GetSortedEdges();
        list<Edge>::iterator it;
        string str = "";
        for (it = edges.begin(); it != edges.end(); it++){
                str += to_string(it->Weight) + " ";
        }
        return str;
```

```
}

template<class T>
void Graph<T>::MarkEntireGraph(bool mark){
	for (int i = 0; i < vertexCount; i++){
		vertices[i].Mark(mark);
	}
	for (int i = 0; i < vertexCount; i++){
		for (int j = 0; j < vertexCount; j++){
			edgeMarkedMatrix[i][j] = false;
		}
	}
}

template<class T>
void Graph<T>::VisitEntireGraph(bool mark){
	for (int i = 0; i < vertexCount; i++){
		vertices[i].Visit(mark);
	}
	for (int i = 0; i < vertexCount; i++){
		for (int j = 0; j < vertexCount; j++){
			edgeVisitedMatrix[i][j] = false;
		}
	}
}

template<class T>
bool Graph<T>::CheckIsConnected() {
	bool result = true;
	bool connected = false;
	for (int i = 0; i < vertexCount; i++){
		connected = false;
		for (int j = 0; j < vertexCount; j++){
			if (adjacencyMatrix[i][j] != 0 || adjacencyMatrix[j][i] != 0){
				connected = true;
				continue;
			}
		}
		if (!connected){
			inTreeMatrix[i] = false;
			result = false;
		}
	}
	return result;
}
```

Pos.h

```
#pragma once
struct Pos{
	int X;
	int Y;
};
```

Main.cpp

```
#include "GraphForm.h"
#include <iostream>
```

```
using namespace GraphProject;
using namespace System;
using namespace System::Windows::Forms;

[STAThreadAttribute]
int main(cli::array<System::String^, 1>^ args) {
	Application::EnableVisualStyles();
	Application::SetCompatibleTextRenderingDefault(false);
	GraphForm ^form = gcnew GraphForm;
	Application::Run(form);

	return 0;
}
```

GraphForm.h

```
#pragma once
#include <ctime>
#include "Graph.h"
#include "Pos.h"

const int ARROW_OFFSET = 30;
const int RANDOM_VERTEX_COUNT = 30;

namespace GraphProject {

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

	protected:
		/// <summary>
		/// Clean up any resources being used.
		/// </summary>
		~GraphForm()
		{
			if (components)
			{
				delete components;
				delete g;
			}
```

```
       }
private: System::Windows::Forms::Label^  label1;
private: System::Windows::Forms::Button^  primButton;
private: System::Windows::Forms::Button^  kruskalButton;
protected:


private: System::Windows::Forms::Label^  label2;
private: System::Windows::Forms::Button^  dijkstraButton;
private: System::Windows::Forms::TextBox^  vertexBox;


private: System::Windows::Forms::Label^  label3;


private: System::Windows::Forms::Label^  label4;
private: System::Windows::Forms::Button^  resetButton;
private: System::Windows::Forms::Button^  breadthButton;
private: System::Windows::Forms::Button^  depthButton;
private: System::Windows::Forms::Label^  label5;
private: System::Windows::Forms::TextBox^  searchVertexBox;
private: System::Windows::Forms::Label^  SearchLabel;
private: System::Windows::Forms::ListBox^  listBox1;
private: System::Windows::Forms::CheckBox^  checkBox1;
private: System::Windows::Forms::TextBox^  dijkstraStart;
private: System::Windows::Forms::TextBox^  dijkstraEnd;


private: System::Windows::Forms::Label^  label6;
private: System::Windows::Forms::Label^  label7;
private: System::Windows::Forms::Button^  undirectedButton;
private: System::Windows::Forms::Button^  acyclicButton;


private: System::Windows::Forms::Button^  cyclicButton;

private: System::Windows::Forms::Label^  label8;
private: System::Windows::Forms::Button^  randomButton;




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

private:
       //Vertices
       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;
		}

		//Edges
		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 MarkLabel(Label ^label, bool mark){
			if (mark){
				label->BackColor = System::Drawing::Color::Blue;
				label->ForeColor = System::Drawing::Color::White;
			}
			else{
				label->BackColor = System::Drawing::Color::LightBlue;
				label->ForeColor = System::Drawing::Color::Black;
			}
		}

		void UpdateLabels(){
			for (int i = 0; i < graph->VertexCount(); i++){
				if (graph->GetVertex(i)->IsMarked()){
					MarkLabel(labels[i], true);
				}
				else{
					MarkLabel(labels[i], false);
				}
			}
		}

		void LoadVertices(Graph<Pos>* g, string filename){
			ifstream file;
			file.open(filename);
			if (file.fail()){
				file.close();
				System::Windows::Forms::Application::Exit();
			}
			while (!file.eof()){
				string n, x, y;
				file >> n >> x >> y;
				if (n == "")
					continue;
				else if (n[0] == 'T'){
					//May not be used
					continue;
				}
```

```
					else{
						Pos p;
						p.X = stoi(x);
						p.Y = stoi(y);
						g->AddVertex(stoi(n), p);
					}
				}
				file.close();
				g->SetupMatrices();
			}

			void LoadRandomVertices(Graph<Pos>* g){
				for (int i = 0; i < RANDOM_VERTEX_COUNT; i++){
					Pos p;
					p.X = 20 + rand() % 630;
					p.Y = 20 + rand() % 630;
					g->AddVertex(i, p);
				}
				g->SetupMatrices();
			}
			/// <summary>
			/// Required designer variable.
			/// </summary>
			System::ComponentModel::Container ^components;

			Graphics ^g;
			Pen ^edgePen;
			Pen ^markedEdgePen;
			cli::array<Label^>^ labels;
			bool graphLoaded = false;
			int labelCount = 0;

			Graph<Pos> *graph = nullptr;

#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->label1 = (gcnew System::Windows::Forms::Label());
				this->primButton = (gcnew System::Windows::Forms::Button());
				this->kruskalButton = (gcnew System::Windows::Forms::Button());
				this->label2 = (gcnew System::Windows::Forms::Label());
				this->dijkstraButton = (gcnew System::Windows::Forms::Button());
				this->vertexBox = (gcnew System::Windows::Forms::TextBox());
				this->label3 = (gcnew System::Windows::Forms::Label());
				this->label4 = (gcnew System::Windows::Forms::Label());
				this->resetButton = (gcnew System::Windows::Forms::Button());
				this->breadthButton = (gcnew System::Windows::Forms::Button());
				this->depthButton = (gcnew System::Windows::Forms::Button());
				this->label5 = (gcnew System::Windows::Forms::Label());
				this->searchVertexBox = (gcnew System::Windows::Forms::TextBox());
				this->SearchLabel = (gcnew System::Windows::Forms::Label());
				this->listBox1 = (gcnew System::Windows::Forms::ListBox());
				this->checkBox1 = (gcnew System::Windows::Forms::CheckBox());
			this->dijkstraStart = (gcnew System::Windows::Forms::TextBox());
			this->dijkstraEnd = (gcnew System::Windows::Forms::TextBox());
			this->label6 = (gcnew System::Windows::Forms::Label());
			this->label7 = (gcnew System::Windows::Forms::Label());
			this->undirectedButton = (gcnew System::Windows::Forms::Button());
			this->acyclicButton = (gcnew System::Windows::Forms::Button());
			this->cyclicButton = (gcnew System::Windows::Forms::Button());
			this->label8 = (gcnew System::Windows::Forms::Label());
			this->randomButton = (gcnew System::Windows::Forms::Button());
			this->SuspendLayout();
			// 
			// panel1
			// 
			this->panel1->Location = System::Drawing::Point(13, 13);
			this->panel1->Name = L"panel1";
			this->panel1->Size = System::Drawing::Size(759, 737);
			this->panel1->TabIndex = 0;
			this->panel1->Paint += gcnew 
System::Windows::Forms::PaintEventHandler(this, &GraphForm::panel1_Paint);
			// 
			// label1
			// 
			this->label1->AutoSize = true;
			this->label1->Location = System::Drawing::Point(973, 64);
			this->label1->Name = L"label1";
			this->label1->Size = System::Drawing::Size(116, 13);
			this->label1->TabIndex = 1;
			this->label1->Text = L"Make a Spanning Tree";
			// 
			// primButton
			// 
			this->primButton->Location = System::Drawing::Point(976, 80);
			this->primButton->Name = L"primButton";
			this->primButton->Size = System::Drawing::Size(107, 23);
			this->primButton->TabIndex = 2;
			this->primButton->Text = L"Prim\'s Algorithm";
			this->primButton->UseVisualStyleBackColor = true;
			this->primButton->Click += gcnew System::EventHandler(this, 
&GraphForm::primButton_Click);
			// 
			// kruskalButton
			// 
			this->kruskalButton->Location = System::Drawing::Point(976, 109);
			this->kruskalButton->Name = L"kruskalButton";
			this->kruskalButton->Size = System::Drawing::Size(107, 23);
			this->kruskalButton->TabIndex = 3;
			this->kruskalButton->Text = L"Kruskal\'s Algorithm";
			this->kruskalButton->UseVisualStyleBackColor = true;
			this->kruskalButton->Click += gcnew System::EventHandler(this, 
&GraphForm::kruskalButton_Click);
			// 
			// label2
			// 
			this->label2->AutoSize = true;
			this->label2->Location = System::Drawing::Point(997, 171);
			this->label2->Name = L"label2";
			this->label2->Size = System::Drawing::Size(68, 13);
			this->label2->TabIndex = 4;
```

```
                    this->label2->Text = L"Make a Path";
                    //
                    // dijkstraButton
                    //
                    this->dijkstraButton->Location = System::Drawing::Point(976, 187);
                    this->dijkstraButton->Name = L"dijkstraButton";
                    this->dijkstraButton->Size = System::Drawing::Size(107, 23);
                    this->dijkstraButton->TabIndex = 5;
                    this->dijkstraButton->Text = L"Dijkstra\'s Algorithm";
                    this->dijkstraButton->UseVisualStyleBackColor = true;
                    this->dijkstraButton->Click += gcnew System::EventHandler(this,
&GraphForm::dijkstraButton_Click);
                    //
                    // vertexBox
                    //
                    this->vertexBox->Location = System::Drawing::Point(933, 80);
                    this->vertexBox->MaxLength = 2;
                    this->vertexBox->Name = L"vertexBox";
                    this->vertexBox->Size = System::Drawing::Size(37, 20);
                    this->vertexBox->TabIndex = 6;
                    //
                    // label3
                    //
                    this->label3->AutoSize = true;
                    this->label3->Location = System::Drawing::Point(891, 64);
                    this->label3->Name = L"label3";
                    this->label3->Size = System::Drawing::Size(76, 13);
                    this->label3->TabIndex = 7;
                    this->label3->Text = L"Starting Vertex";
                    //
                    // label4
                    //
                    this->label4->AutoSize = true;
                    this->label4->Location = System::Drawing::Point(893, 324);
                    this->label4->Name = L"label4";
                    this->label4->Size = System::Drawing::Size(139, 13);
                    this->label4->TabIndex = 9;
                    this->label4->Text = L"Spanning Tree/Path Output";
                    //
                    // resetButton
                    //
                    this->resetButton->Location = System::Drawing::Point(1008, 278);
                    this->resetButton->Name = L"resetButton";
                    this->resetButton->Size = System::Drawing::Size(75, 23);
                    this->resetButton->TabIndex = 10;
                    this->resetButton->Text = L"Reset";
                    this->resetButton->UseVisualStyleBackColor = true;
                    this->resetButton->Click += gcnew System::EventHandler(this,
&GraphForm::resetButton_Click);
                    //
                    // breadthButton
                    //
                    this->breadthButton->Location = System::Drawing::Point(1000, 646);
                    this->breadthButton->Name = L"breadthButton";
                    this->breadthButton->Size = System::Drawing::Size(109, 23);
                    this->breadthButton->TabIndex = 11;
                    this->breadthButton->Text = L"Breadth-first Search";
                    this->breadthButton->UseVisualStyleBackColor = true;
                    this->breadthButton->Click += gcnew System::EventHandler(this, 
&GraphForm::breadthButton_Click);
                    // 
                    // depthButton
                    // 
                    this->depthButton->Location = System::Drawing::Point(1000, 617);
                    this->depthButton->Name = L"depthButton";
                    this->depthButton->Size = System::Drawing::Size(109, 23);
                    this->depthButton->TabIndex = 12;
                    this->depthButton->Text = L"Depth-first Search";
                    this->depthButton->UseVisualStyleBackColor = true;
                    this->depthButton->Click += gcnew System::EventHandler(this, 
&GraphForm::depthButton_Click);
                    // 
                    // label5
                    // 
                    this->label5->AutoSize = true;
                    this->label5->Location = System::Drawing::Point(989, 594);
                    this->label5->Name = L"label5";
                    this->label5->Size = System::Drawing::Size(76, 13);
                    this->label5->TabIndex = 7;
                    this->label5->Text = L"Starting Vertex";
                    // 
                    // searchVertexBox
                    // 
                    this->searchVertexBox->Location = System::Drawing::Point(1072, 591);
                    this->searchVertexBox->MaxLength = 2;
                    this->searchVertexBox->Name = L"searchVertexBox";
                    this->searchVertexBox->Size = System::Drawing::Size(37, 20);
                    this->searchVertexBox->TabIndex = 6;
                    // 
                    // SearchLabel
                    // 
                    this->SearchLabel->BackColor = 
System::Drawing::SystemColors::ButtonHighlight;
                    this->SearchLabel->Location = System::Drawing::Point(781, 692);
                    this->SearchLabel->Name = L"SearchLabel";
                    this->SearchLabel->Size = System::Drawing::Size(328, 65);
                    this->SearchLabel->TabIndex = 13;
                    // 
                    // listBox1
                    // 
                    this->listBox1->FormattingEnabled = true;
                    this->listBox1->Location = System::Drawing::Point(781, 340);
                    this->listBox1->Name = L"listBox1";
                    this->listBox1->Size = System::Drawing::Size(328, 225);
                    this->listBox1->TabIndex = 14;
                    // 
                    // checkBox1
                    // 
                    this->checkBox1->AutoSize = true;
                    this->checkBox1->Location = System::Drawing::Point(947, 138);
                    this->checkBox1->Name = L"checkBox1";
                    this->checkBox1->Size = System::Drawing::Size(142, 17);
                    this->checkBox1->TabIndex = 15;
                    this->checkBox1->Text = L"Show only spanning tree";
                    this->checkBox1->UseVisualStyleBackColor = true;
```

```
			this->checkBox1->CheckedChanged += gcnew System::EventHandler(this, 
&GraphForm::checkBox1_CheckedChanged);
			// 
			// dijkstraStart
			// 
			this->dijkstraStart->Location = System::Drawing::Point(893, 187);
			this->dijkstraStart->MaxLength = 2;
			this->dijkstraStart->Name = L"dijkstraStart";
			this->dijkstraStart->Size = System::Drawing::Size(37, 20);
			this->dijkstraStart->TabIndex = 6;
			// 
			// dijkstraEnd
			// 
			this->dijkstraEnd->Location = System::Drawing::Point(936, 187);
			this->dijkstraEnd->MaxLength = 2;
			this->dijkstraEnd->Name = L"dijkstraEnd";
			this->dijkstraEnd->Size = System::Drawing::Size(37, 20);
			this->dijkstraEnd->TabIndex = 6;
			// 
			// label6
			// 
			this->label6->AutoSize = true;
			this->label6->Location = System::Drawing::Point(893, 171);
			this->label6->Name = L"label6";
			this->label6->Size = System::Drawing::Size(29, 13);
			this->label6->TabIndex = 7;
			this->label6->Text = L"Start";
			// 
			// label7
			// 
			this->label7->AutoSize = true;
			this->label7->Location = System::Drawing::Point(938, 171);
			this->label7->Name = L"label7";
			this->label7->Size = System::Drawing::Size(26, 13);
			this->label7->TabIndex = 7;
			this->label7->Text = L"End";
			// 
			// undirectedButton
			// 
			this->undirectedButton->Location = System::Drawing::Point(781, 31);
			this->undirectedButton->Name = L"undirectedButton";
			this->undirectedButton->Size = System::Drawing::Size(102, 23);
			this->undirectedButton->TabIndex = 16;
			this->undirectedButton->Text = L"Undirected Graph";
			this->undirectedButton->UseVisualStyleBackColor = true;
			this->undirectedButton->Click += gcnew System::EventHandler(this, 
&GraphForm::undirectedButton_Click);
			// 
			// acyclicButton
			// 
			this->acyclicButton->Location = System::Drawing::Point(1015, 31);
			this->acyclicButton->Name = L"acyclicButton";
			this->acyclicButton->Size = System::Drawing::Size(102, 23);
			this->acyclicButton->TabIndex = 16;
			this->acyclicButton->Text = L"Acyclic Graph";
			this->acyclicButton->UseVisualStyleBackColor = true;
			this->acyclicButton->Click += gcnew System::EventHandler(this, 
&GraphForm::acyclicButton_Click);
```

```
                    //
                    // cyclicButton
                    //
                    this->cyclicButton->Location = System::Drawing::Point(898, 31);
                    this->cyclicButton->Name = L"cyclicButton";
                    this->cyclicButton->Size = System::Drawing::Size(102, 23);
                    this->cyclicButton->TabIndex = 16;
                    this->cyclicButton->Text = L"Cyclic Graph";
                    this->cyclicButton->UseVisualStyleBackColor = true;
                    this->cyclicButton->Click += gcnew System::EventHandler(this,
&GraphForm::cyclicButton_Click);
                    //
                    // label8
                    //
                    this->label8->AutoSize = true;
                    this->label8->Location = System::Drawing::Point(922, 15);
                    this->label8->Name = L"label8";
                    this->label8->Size = System::Drawing::Size(72, 13);
                    this->label8->TabIndex = 17;
                    this->label8->Text = L"Load a Graph";
                    //
                    // randomButton
                    //
                    this->randomButton->Location = System::Drawing::Point(781, 64);
                    this->randomButton->Name = L"randomButton";
                    this->randomButton->Size = System::Drawing::Size(102, 23);
                    this->randomButton->TabIndex = 16;
                    this->randomButton->Text = L"Random Graph";
                    this->randomButton->UseVisualStyleBackColor = true;
                    this->randomButton->Click += gcnew System::EventHandler(this,
&GraphForm::randomButton_Click);
                    //
                    // GraphForm
                    //
                    this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
                    this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
                    this->ClientSize = System::Drawing::Size(1129, 762);
                    this->Controls->Add(this->label8);
                    this->Controls->Add(this->randomButton);
                    this->Controls->Add(this->cyclicButton);
                    this->Controls->Add(this->acyclicButton);
                    this->Controls->Add(this->undirectedButton);
                    this->Controls->Add(this->checkBox1);
                    this->Controls->Add(this->listBox1);
                    this->Controls->Add(this->SearchLabel);
                    this->Controls->Add(this->depthButton);
                    this->Controls->Add(this->breadthButton);
                    this->Controls->Add(this->resetButton);
                    this->Controls->Add(this->label4);
                    this->Controls->Add(this->label5);
                    this->Controls->Add(this->label7);
                    this->Controls->Add(this->label6);
                    this->Controls->Add(this->label3);
                    this->Controls->Add(this->searchVertexBox);
                    this->Controls->Add(this->dijkstraEnd);
                    this->Controls->Add(this->dijkstraStart);
                    this->Controls->Add(this->vertexBox);
                    this->Controls->Add(this->dijkstraButton);
```

```
			this->Controls->Add(this->label2);
			this->Controls->Add(this->kruskalButton);
			this->Controls->Add(this->primButton);
			this->Controls->Add(this->label1);
			this->Controls->Add(this->panel1);
			this->Name = L"GraphForm";
			this->Text = L"Graph";
			this->Load += gcnew System::EventHandler(this,
&GraphForm::GraphForm_Load);
			this->ResumeLayout(false);
			this->PerformLayout();

		}

		void LoadGraph(string graphName){
			if (graph != nullptr){
				delete graph;
			}
			graph = new Graph<Pos>();
			LoadVertices(graph, graphName + " Graph Vertices.txt");
			graph->LoadEdges(graphName + " Graph Edges.txt");
			labels = gcnew cli::array<Label^>(graph->VertexCount());

			for (int i = 0; i < labelCount; i++){
				panel1->Controls->Remove(labels[i]);
			}
			labelCount = 0;

			//Display edge weights
			//O(n^2)
			for (int i = 0; i < graph->VertexCount(); i++){
				Label^ l = newLabel(i.ToString(), graph->GetVertex(i)-
>GetT().X, graph->GetVertex(i)->GetT().Y);
				labels[labelCount] = l;
				labelCount++;

				for (int j = 0; j < graph->VertexCount(); j++){
					int w = graph->GetEdgeWeight(i, j);
					if (w == 0)
						continue;
					Pos p1 = graph->GetVertex(i)->GetT();
					Pos p2 = graph->GetVertex(j)->GetT();
					int x1 = p1.X;
					int y1 = p1.Y;
					int x2 = p2.X;
					int y2 = p2.Y;
					Point center = Point(x1 + (x2 - x1) / 2, y1 + (y2 - y1)
/ 2);

					newLabel2(w.ToString(), center.X + 8, center.Y + 8);
				}
			}
			EnableGraphButtons(false);
			graphLoaded = true;
			panel1->Refresh();
		}

		void LoadGraphRandom(){
			if (graph != nullptr){
				delete graph;
			}
			graph = new Graph<Pos>();
			LoadRandomVertices(graph);
			graph->LoadRandomEdges();
			labels = gcnew cli::array<Label^>(graph->VertexCount());

			for (int i = 0; i < graph->VertexCount(); i++){
				Label^ l = newLabel(i.ToString(), graph->GetVertex(i)-
>GetT().X, graph->GetVertex(i)->GetT().Y);
				labels[labelCount] = l;
				labelCount++;

				for (int j = 0; j < graph->VertexCount(); j++){
					int w = graph->GetEdgeWeight(i, j);
					if (w == 0)
						continue;
					Pos p1 = graph->GetVertex(i)->GetT();
					Pos p2 = graph->GetVertex(j)->GetT();
					int x1 = p1.X;
					int y1 = p1.Y;
					int x2 = p2.X;
					int y2 = p2.Y;
					Point center = Point(x1 + (x2 - x1) / 2, y1 + (y2 - y1)
/ 2);

					newLabel2(w.ToString(), center.X + 8, center.Y + 8);
				}
			}
			EnableGraphButtons(false);
			graphLoaded = true;
			panel1->Refresh();
		}

		void EnableButtons(bool enabled){
			primButton->Enabled = enabled;
			kruskalButton->Enabled = enabled;
			dijkstraButton->Enabled = enabled;
			resetButton->Enabled = enabled;
			breadthButton->Enabled = enabled;
			depthButton->Enabled = enabled;
		}

		void EnableGraphButtons(bool enabled){
			undirectedButton->Enabled = enabled;
			cyclicButton->Enabled = enabled;
			acyclicButton->Enabled = enabled;
		}

#pragma endregion
	private: System::Void GraphForm_Load(System::Object^  sender, System::EventArgs^
e) {
		srand(time(0));
		graph = nullptr;
		labelCount = 0;
		g = panel1->CreateGraphics();
		edgePen = gcnew Pen(Color::LightBlue, 6);
		edgePen->EndCap = Drawing2D::LineCap::ArrowAnchor;
		markedEdgePen = gcnew Pen(Color::Blue, 6);
```

```
			markedEdgePen->EndCap = Drawing2D::LineCap::ArrowAnchor;
			bool result = false;

			EnableButtons(false);
		}
		private: System::Void panel1_Paint(System::Object^  sender, 
System::Windows::Forms::PaintEventArgs^  e) {
			if (!graphLoaded)
				return;
			//Display edges
			for (int i = 0; i < graph->VertexCount(); i++){
				for (int j = 0; j < graph->VertexCount(); j++){
					int w = graph->GetEdgeWeight(i, j);
					if (w == 0)
						continue;
					Vertex<Pos> *v1 = graph->GetVertex(i);
					Vertex<Pos> *v2 = graph->GetVertex(j);
					int x1_ = v1->GetT().X;
					int y1_ = v1->GetT().Y;
					int x2_ = v2->GetT().X;
					int y2_ = v2->GetT().Y;

					float direction = Math::Atan2(y2_ - y1_, x2_ - x1_);

					int x1 = x1_ + (ARROW_OFFSET * Math::Cos(direction));
					int y1 = y1_ + (ARROW_OFFSET * Math::Sin(direction));
					int x2 = x2_ - (ARROW_OFFSET * Math::Cos(direction));
					int y2 = y2_ - (ARROW_OFFSET * Math::Sin(direction));

					if (!graph->GetEdgeMarked(i, j)){
						if (!checkBox1->Checked){
							g->DrawLine(edgePen, x1 + 16, y1 + 16, x2 + 16, 
y2 + 16);
						}
					}
					else{
						g->DrawLine(markedEdgePen, x1 + 16, y1 + 16, x2 + 16, 
y2 + 16);
					}
				}
			}
		}
		private: System::Void primButton_Click(System::Object^  sender, System::EventArgs^ 
e) {
			listBox1->Items->Clear();
			ListBox::ObjectCollection ^col = gcnew ListBox::ObjectCollection(listBox1);
			int num = -1;
			bool result = int::TryParse(vertexBox->Text, num);
			if (!result || num < 0 || num >= graph->VertexCount())
				return;

			graph->Prim(num, col);
			UpdateLabels();
			panel1->Refresh();
		}
		private: System::Void resetButton_Click(System::Object^  sender, 
System::EventArgs^  e) {
			listBox1->Items->Clear();
```

```
                graph->MarkEntireGraph(false);
                UpdateLabels();
                panel1->Refresh();
        }
private: System::Void kruskalButton_Click(System::Object^  sender, System::EventArgs^  e)
{
        listBox1->Items->Clear();
        ListBox::ObjectCollection ^col = gcnew ListBox::ObjectCollection(listBox1);
        graph->Kruskal(col);
        UpdateLabels();
        panel1->Refresh();
}
private: System::Void depthButton_Click(System::Object^  sender, System::EventArgs^  e) {
        int num = -1;
        String ^result = gcnew String("");
        bool success = int::TryParse(searchVertexBox->Text, num);
        if (!success || num < 0 || num >= graph->VertexCount())
                return;

        graph->DepthFirstSearch(num, result);

        SearchLabel->Text = result;

        graph->VisitEntireGraph(false);
        graph->MarkEntireGraph(false);
        UpdateLabels();

        panel1->Refresh();
}
private: System::Void breadthButton_Click(System::Object^  sender, System::EventArgs^  e)
{
        graph->MarkEntireGraph(false);
        int num = -1;
        bool success = int::TryParse(searchVertexBox->Text, num);
        if (!success || num < 0 || num >= graph->VertexCount())
                return;

        String ^result = gcnew String(num + " ");
        graph->GetVertex(num)->Mark(true);

        graph->BreadthFirstSearch(num, result);

        SearchLabel->Text = result;

        graph->VisitEntireGraph(false);
        graph->MarkEntireGraph(false);
        UpdateLabels();

        panel1->Refresh();
}
private: System::Void checkBox1_CheckedChanged(System::Object^  sender,
System::EventArgs^  e) {
        panel1->Refresh();
}
        private: System::Void dijkstraButton_Click(System::Object^  sender,
System::EventArgs^  e) {
                listBox1->Items->Clear();
                int start = -1, end = -1;
```

```
              bool success = int::TryParse(dijkstraStart->Text, start);
              if (!success || start < 0 || start >= graph->VertexCount())
                     return;

              success = int::TryParse(dijkstraEnd->Text, end);
              if (!success || end < 0 || end >= graph->VertexCount())
                     return;

              bool pathResult = graph->Dijkstra(start, end);
              if (!pathResult)
                     listBox1->Items->Add("Unable to construct a path from " + start + "
to " + end);
              UpdateLabels();
              panel1->Refresh();
       }
private: System::Void cyclicButton_Click(System::Object^  sender, System::EventArgs^  e)
{
       LoadGraph("Cyclic");
       EnableButtons(true);
}
private: System::Void undirectedButton_Click(System::Object^  sender, System::EventArgs^
e) {
       LoadGraph("Small");
       EnableButtons(true);
}
private: System::Void acyclicButton_Click(System::Object^  sender, System::EventArgs^  e)
{
       LoadGraph("Acyclic");
       EnableButtons(true);
}
private: System::Void randomButton_Click(System::Object^  sender, System::EventArgs^  e)
{
       LoadGraphRandom();
       EnableButtons(true);
}
};
}
```