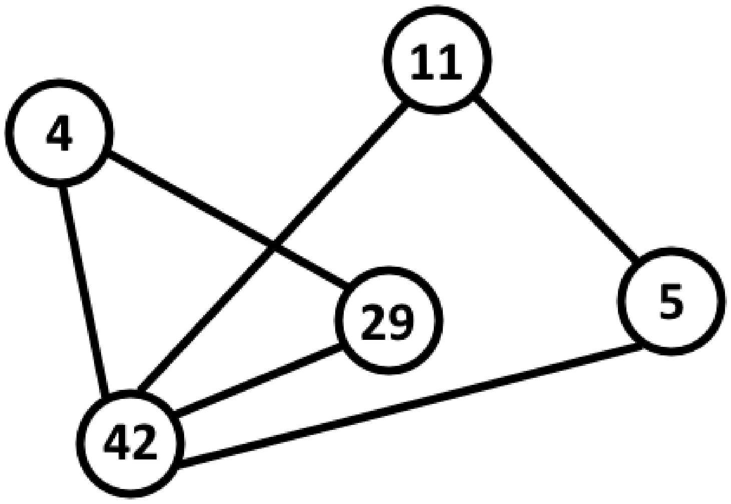
Graphs Reading

Tim "Dr. T" Chamillard

What's a Graph?

A graph is a set of nodes (also called *vertices* or *vertexes*) connected by edges; the picture below shows one example.



There are actually lots of kinds of graphs. The graph shown above is *undirected*, which means edges can be followed in either direction. Graphs can also be *directed*, where each edge can only be followed in one direction. The edges in a graph can be *unweighted* (as shown above), where following one edge "costs" the same as following any other edge. Edges can also be *weighted*, where each edge has a specific cost for following it. Finally, a graph can be *acyclic* (there's no way to get back to a node once you leave it) or a graph can contain one or more cycles.

So what are graphs good for? They're obviously great at representing how nodes are connected to each other, but so what? Let's start with a common, non-game related example. Thinks of a set of cities as nodes in a graph with the roads connecting the cities as the edges in the graph. In this case, we can weight each edge with the length of the edge (the distance you'd have to travel to follow the edge from start to finish). Once we've done that, you could write code to find the shortest path between any two nodes in the graph. You can also solve the Traveling Salesman Problem, where you need to visit each node (city) in the graph exactly once and return to the city you started at for the least possible cost. There are lots of useful problems we can solve traversing graphs.

Sure, but what about for games? One useful thing to do in a game is to support pathfinding by connecting particular locations in the game with edges; if we want to include information about the difficulty of traversing the edge (because of rivers, mountains, etc), we can weight those edges as well. Our NPCs could then use appropriate graph traversal logic to optimize their movement from point to point in the game world. In addition, scene graphs are commonly used in computer graphics. You'll probably come across other examples as well as you develop games.

There are a variety of ways to actually implement graphs depending on whether or not they're *sparse* (not many connections from each node to other nodes) or *dense* (each node is connected to all or most of the other nodes in the graph) and on other factors as well. In this reading, we'll use something called the *adjacency list* approach, where each node holds a list of its neighbors; this is the preferred structure for sparse graphs. We'll implement an undirected, unweighted graph in this reading.

Because there are a variety of factors determining the best implementation of a graph data structure that meets your specific needs, C# doesn't provide a `Graph` class. If you need one, you need to write one! That's okay; we'll do that now.

Graph Node

Let's start with the `GraphNode` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Graphs
{
    /// <summary>
    /// A graph node
    /// </summary>
    /// <typeparam name="T">type of value stored in node</typeparam>
    class GraphNode<T>
    {
```

We're making our node a generic so it can contain any data type for the node value.

```
        #region Fields

        T value;
        List<GraphNode<T>> neighbors;

        #endregion
```

Each node contains a value and a list of the other nodes it's connected to.

```
        #region Constructors

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="value">value for the node</param>
        public GraphNode(T value)
        {
            this.value = value;
            neighbors = new List<GraphNode<T>>();
        }

        #endregion
```

The constructor is pretty straightforward, initializing the value and creating a new (empty) list of neighbors.

```
        #region Properties

        /// <summary>
        /// Gets the value stored in the node
        /// </summary>
        public T Value
        {
            get { return value; }
        }
```

The `Value` property is read-only, so we can get the node value but we can't change it after the node has been created.

```
        /// <summary>
        /// Gets a read-only list of the neighbors of the node
        /// </summary>
        public IList<GraphNode<T>> Neighbors
        {
            get { return neighbors.AsReadOnly(); }
        }

        #endregion
```

The `Neighbors` property provides read-only access to the node's neighbors, but we're using some ideas we haven't discussed yet. That's because if we return a `List` from the property, the consumer of the class can change the contents of the list however they want to. It's even worse than that, though; because `List` is a reference type, the changes they make to their copy will also happen in our graph! We could copy the list of neighbors into a new list before returning it, but there's a better way in C#, at least in this case.

We start by returning `neighbors.AsReadOnly()` instead of `neighbors`. The documentation for the `AsReadOnly` method tells us that the method returns a `ReadOnlyCollection` object that doesn't expose methods that modify the collection. Because `List` isn't a `ReadOnlyCollection`, the property returns an `IList` (an object that implements the `IList` interface) instead. Because `ReadOnlyCollection` implements that interface, this works fine.

The consumer of the class can put the value the property returns into an `IList` and use any of the `IList` methods without any trouble; they just won't be able to modify the contents of that list. Of course, they could still make changes to the nodes in that list (adding and removing neighbors, for example) and those changes would be propagated into our graph because `GraphNode` is also a reference type. We'll say that for this problem we've added enough defenses to be satisfied, but if this turned into a problem in practice we could always do a "deep copy" of the neighbors, making a copy of each of the neighbor nodes to put into a copy of the neighbors list to return to the consumer of the class.

```
       #region Methods

        /// <summary>
        /// Adds the given node as a neighbor for this node
        /// </summary>
        /// <param name="neighbor">neighbor to add</param>
        /// <returns>true if the neighbor was added,
        ///     false otherwise</returns>
        public bool AddNeighbor(GraphNode<T> neighbor)
        {
            // don't add duplicate nodes
            if (neighbors.Contains(neighbor))
            {
                return false;
            }
            else
            {
                neighbors.Add(neighbor);
                return true;
            }
        }
```

The `AddNeighbor` method adds a node to the neighbors for the node as long as that node isn't already a neighbor. If we had weighted edges, allowing multiple edges between two nodes could be useful, but because the edges are unweighted in our graph we only need one edge to connect two nodes in the graph.

The `List Contains` method performs an O(n) linear search, so the AddNeighbor operation is O(n) where n is the number of neighbors.

```
        /// <summary>
        /// Removes the given node as a neighbor for this node
        /// </summary>
        /// <param name="neighbor">neighbor to remove</param>
        /// <returns>true if the neighbor was removed,
        ///     false otherwise</returns>
        public bool RemoveNeighbor(GraphNode<T> neighbor)
        {
            // only remove neighbors in list
            return neighbors.Remove(neighbor);
        }
```

We're taking advantage of the fact that the `List Remove` method returns false if the neighbor we're trying to remove isn't in the list.

The `List Remove` method performs an O(n) linear search for the item to remove, so the RemoveNeighbor operation is O(n) where n is the number of neighbors.

```
        /// <summary>
        /// Removes all the neighbors for the node
        /// </summary>
        /// <returns>true if the neighbors were removed,
        ///     false otherwise</returns>
        public bool RemoveAllNeighbors()
        {
            for (int i = neighbors.Count - 1; i >= 0; i--)
            {
                neighbors.RemoveAt(i);
            }
            return true;
        }
```

The `RemoveAllNeighbors` method removes all the neighbors for the given node. This is useful when we want to clear a graph of nodes because removing the reference from a node to its neighbors makes it so the garbage collector can collect the nodes.

The `List RemoveAt` method is O(Count - index) where index is the location we want to remove. We're always passing in an index of Count - 1 (because we're going backwards through the list), so the body of the loop is O(1) because Count - (Count - 1) is 1. The loop executes n times, where n is the number of neighbors, so the RemoveAllNeighbors operation is O(n).

```
        /// <summary>
        /// Converts the node to a string
        /// </summary>
        /// <returns>the string</returns>
        public override string ToString()
        {
            StringBuilder nodeString = new StringBuilder();
            nodeString.Append("[Node Value: " + value +
                " Neighbors: ");
            for (int i = 0; i < neighbors.Count; i++)
            {
                nodeString.Append(neighbors[i].Value + " ");
            }
            nodeString.Append("]");
            return nodeString.ToString();
        }

        #endregion
    }
}
```

I implemented the `ToString` method to support the test cases, but being able to get a string representation of the node could be useful in other scenarios as well. The ToString operation is clearly O(n) because the for loop executes n times where n is the number of neighbors.

The `TestGraphNode` class in the code accompanying this reading runs a variety of tests against the code above. Feel free to take a look at the test cases if you'd like.

Graph

Okay, on to the `Graph` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Graphs
{
    /// <summary>
    /// A graph
    /// </summary>
    /// <typeparam name="T">type of values stored in graph</typeparam>
    class Graph<T>
    {
        #region Fields

        List<GraphNode<T>> nodes = new List<GraphNode<T>>();

        #endregion
```

Our `Graph` class is also a generic, of course. The only field we have holds a list of the nodes in the graph. Notice that a graph doesn't have a head like a linked list does; there's no set starting point in a graph, though you can of course traverse a graph starting at any node.

```
        #region Constructor

        /// <summary>
        /// Constructor
        /// </summary>
        public Graph()
        {
        }

        #endregion
```

Our constructor doesn't actually do anything, but I prefer to explicitly include the no argument constructor in my code.

```
        #region Properties

        /// <summary>
        /// Gets the number of nodes in the graph
        /// </summary>
        public int Count
        {
            get { return nodes.Count; }
        }

        /// <summary>
        /// Gets a read-only list of the nodes in the graph
        /// </summary>
        public IList<GraphNode<T>> Nodes
        {
            get { return nodes.AsReadOnly(); }
        }

        #endregion
```

We provide a `Count` property to get the number of nodes in the graph and the `Nodes` property provides a read-only list of the nodes in the graph.

```
        /// <summary>
        /// Clears all the nodes from the graph
        /// </summary>
        public void Clear()
        {
            // remove all the neighbors from each node
            // so nodes can be garbage collected
            foreach (GraphNode<T> node in nodes)
            {
                node.RemoveAllNeighbors();
            }

            // now remove all the nodes from the graph
            for (int i = nodes.Count - 1; i >= 0; i--)
            {
                nodes.RemoveAt(i);
            }
        }
```

As the comment says, we want to unlink all the neighbor nodes for each node so the garbage collector can collect them. After doing that, we remove all the nodes from the graph.

Graph complexities are typically expressed in terms of the number of Vertices (nodes) in the graph, V, and the number of Edges in the graph, E. The complexity of the second loop is clearly O(V), where V is the number of nodes in the graph, but the first for loop requires a little more analysis. The loop clearly executes V times, but what's the complexity of the loop body? When we analyzed the `GraphNode` `RemoveAllNeighbors` method, we determined that operation is O(n) where n is the number of neighbors for the node. That's different from the number of nodes in the graph, so how can we reconcile that? Well, in the worst case, we have a graph in which every node is connected to every other node (this is called a *complete graph*). In this case, every node has V - 1 neighbors, so the RemoveAllNeighbors operation is O(V).

Remember that we need to multiply the complexity of the loop body by the number of iterations the loop performs to calculate the complexity of the loop. For the first loop, the loop body has O(V) complexity and the loop executes V - 1 times in the worst case, so the complexity of the first loop is O(V^2). Because the first loop is O(V^2) and the second loop is O(V), the Clear operation is O(V^2) where V is the number of vertices (nodes) in the graph.

This analysis is fine for all kinds of graphs, but it's most accurate for dense graphs, where most or all of the nodes are connected to each other with edges. In a sparse graph, our worst-case assumption that each node is connected to all of the other nodes in the graph when we remove the node's neighbors is far worse than it is in practice. To address this, we can include E, the number of edges in the graph, in our analysis. The Clear operation then becomes O(V * E), because the first loop executes V times but the body of that loop only removes 2E neighbors for the entire graph (because each edge has a neighbor entry in both nodes at the ends of the edge).

```
        /// <summary>
        /// Adds a node with the given value to the graph. If a node
        /// with the same value is in the graph, the value
        /// isn't added and the method returns false
        /// </summary>
        /// <param name="value">value to add</param>
        /// <returns>true if the value is added, false otherwise</returns>
        public bool AddNode(T value)
        {
            if (Find(value) != null)
            {
                // duplicate value
                return false;
            }
            else
            {
                nodes.Add(new GraphNode<T>(value));
                return true;
            }
        }
```

The `AddNode` method adds a node to the graph if a node with that value isn't in the graph already. It would certainly be reasonable to allow multiple nodes with the same value in some scenarios, but if we think of each node as a location in the game world to support pathfinding, multiple nodes with the same value don't make sense. Using a graph to capture distinct narrative points a player can reach is reasonable for a non-linear story, but a particular narrative point (perhaps expressed through a cut scene) would be unique in the narrative graph as well. We'll stick with unique node values in our implementation here.

We'll discover when we get to it that the `Find` method is O(V). Because the `List Add` method is O(V) in the worst case (when we need to grow the list), AddNode is an O(V) operation where V is the number of vertices (nodes) in the graph.

```
        /// <summary>
        /// Adds an edge between the nodes with the given values
        /// in the graph. If one or both of the values don't exist
        /// in the graph the method returns false. If an edge
        /// already exists between the nodes the edge isn't added
        /// and the method retruns false
        /// </summary>
        /// <param name="value1">first value to connect</param>
        /// <param name="value2">second value to connect</param>
        /// <returns>true if the edge is added, false otherwise</returns>
        public bool AddEdge(T value1, T value2)
        {
            GraphNode<T> node1 = Find(value1);
            GraphNode<T> node2 = Find(value2);
            if (node1 == null ||
                node2 == null)
            {
                return false;
            }
            else if (node1.Neighbors.Contains(node2))
            {
                // edge already exists
                return false;
            }
            else
            {
                // undirected graph, so add as neighbors to each other
                node1.AddNeighbor(node2);
                node2.AddNeighbor(node1);
                return true;
            }
        }
```

The `AddEdge` method adds an edge between two nodes in the graph as long as both nodes exist in the graph and the edge between the nodes doesn't. It should be obvious why we can't connect graph nodes that don't exist; we don't allow duplicate edges because one edge between two nodes is sufficient to connect them. If we actually are going to add the edge, we add each node as a neighbor to the other node.

There are situations where you might want to allow multiple edges between two nodes. If the edges are weighted, you could have edges with different weights between two nodes. If we return to our cities example, we'd probably have different distances between two cities based on the route taken. If we were weighting the edges with airline ticket costs, edges could have different weights (ticket prices) based on the time of day. We'll stick with unique edges in our implementation, but there are plenty of times when multiple edges between two nodes makes sense.

The calls to the `Find` method at the top of the method are both O(V). The `List Contains` method, which we use to see if there's already an edge between node1 and node2, is O(V) where n is the number of neighbors node1 has (which is V - 1 in the worst case). The `GraphNode AddNeighbor` method is O(V) where n is the number of neighbors the node has (which is V - 1 in the worst case). When we consider all those complexities (which are added, not multiplied) we see that AddEdge is an O(V) operation where V is the number of vertices (nodes) in the graph.

```
        /// <summary>
        /// Removes the node with the given value from the graph.
        /// If the node isn't found in the graph, the method
        /// returns false
        /// </summary>
        /// <param name="value">value to remove</param>
        /// <returns>true if the value is removed, false otherwise</returns>
        public bool RemoveNode(T value)
        {
            GraphNode<T> removeNode = Find(value);
            if (removeNode == null)
            {
                return false;
            }
            else
            {
                // need to remove as neighbor for all nodes
                // in graph
                nodes.Remove(removeNode);
                foreach (GraphNode<T> node in nodes)
                {
                    node.RemoveNeighbor(removeNode);
                }
                return true;
            }
        }
```

The `RemoveNode` method removes the node with the given value from the graph. To do that, it also has to remove the node from the list of neighbors for all the other nodes in the graph.

The call to the `Find` method at the top of the method is O(V) and the `List Remove` method is also O(V). The `GraphNode RemoveNeighbor` method is O(V) where n is the number of neighbors for the node, which is V - 1 nodes in the worst case. Because that O(V) operation is contained in a loop that executes V times, RemoveNode is an O(V^2) operation where V is the number of vertices (nodes) in the graph.

```
        /// <summary>
        /// Removes an edge between the nodes with the given values
        /// from the graph. If one or both of the values don't exist
        /// in the graph the method returns false
        /// </summary>
        /// <param name="value1">first value to disconnect</param>
        /// <param name="value2">second value to disconnect</param>
        /// <returns>true if the edge is removed, false otherwise</returns>
        public bool RemoveEdge(T value1, T value2)
        {
            GraphNode<T> node1 = Find(value1);
            GraphNode<T> node2 = Find(value2);
            if (node1 == null ||
                node2 == null)
            {
                return false;
            }
            else if (!node1.Neighbors.Contains(node2))
            {
                // edge doesn't exist
                return false;
            }
            else
            {
                // undirected graph, so remove as neighbors to each other
                node1.RemoveNeighbor(node2);
                node2.RemoveNeighbor(node1);
                return true;
            }
        }
```

The `RemoveEdge` method removes an edge between two nodes as long as both nodes exist in the graph and the edge between the nodes exists. It should be obvious why we can't disconnect graph nodes that don't exist or remove an edge that doesn't exist. If we actually are going to remove the edge, we remove each node as a neighbor for the other node.

The calls to the `Find` method at the top of the method are both O(V). The `List Contains` method, which we use to see if the edge exists between node1 and node2, is O(V) where n is the number of neighbors node1 has (which is V - 1 in the worst case). The `GraphNode RemoveNeighbor` method is O(V) where n is the number of neighbors the node has (which is V - 1 in the worst case). When we consider all those complexities (which are added, not multiplied) we see that RemoveEdge is an O(V) operation where V is the number of vertices (nodes) in the graph.

```
        /// <summary>
        /// Finds the graph node with the given value
        /// </summary>
        /// <param name="value">value to find</param>
        /// <returns>graph node or null if not found</returns>
        public GraphNode<T> Find(T value)
        {
            foreach (GraphNode<T> node in nodes)
            {
                if (node.Value.Equals(value))
                {
                    return node;
                }
            }
            return null;
        }
```

The `Find` method walks the list of nodes in the graph, returning the node with the given value if it finds it. If it doean't find the value in the graph, it returns null.

The loop executes V times, so Find is an O(V) operation where V is the number of vertices (nodes) in the graph.

```
        /// <summary>
        /// Converts the Graph to a comma-separated string of nodes
        /// </summary>
        /// <returns>comma-separated string of nodes</returns>
        public override String ToString()
        {
            StringBuilder builder = new StringBuilder();
            for (int i = 0; i < Count; i++)
            {
                builder.Append(nodes[i].ToString());
                if (i < Count - 1)
                {
                    builder.Append(",");
                }
            }
            return builder.ToString();
        }

        #endregion
    }
}
```

I implemented the `ToString` method to support the test cases, but being able to get a string representation of the graph could be useful in other scenarios as well.

The ToString operation is clearly O(V^2) because the for loop executes V times where V is the number of vertices (nodes) in the graph and, inside the loop, the `GraphNode ToString` method is O(V) where n is the number of neighbors the node has (which is V - 1 in the worst case).

The `TestGraph` class in the code accompanying this reading runs a variety of tests against the code above. Feel free to take a look at the test cases if you'd like.