Dijkstra's Algorithm

.NET 3.5+

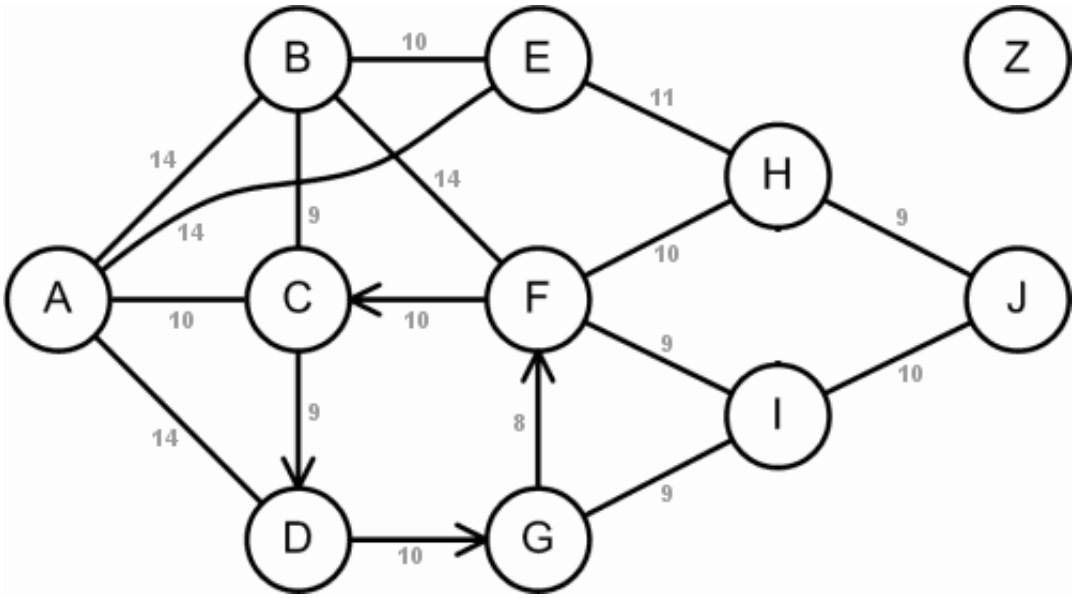
Graph traversal algorithms are used to process a graph of interconnected nodes, visiting each node by following a set behaviour. Dijkstra's algorithm is such an algorithm. It determines the shortest path from a start point to all other nodes in a graph.

Download Demo and Source Code

Graph Traversal

In mathematics, a *graph* is a set of interconnected nodes. These nodes often represent real-world objects. For example, a graph may be used to represent a road network, with each node corresponding to a city, town, village or other place of interest. The connections between the nodes would represent the roads that join those places. A graph's connections may be *symmetric* or *asymmetric*, also known as *undirected* or *directed* respectively. A symmetric connection can be followed in both directions, such as most roads. An asymmetric connection can only be followed in one direction, perhaps representing a one-way street.

The diagram below shows an abstract representation of a graph. In this case we have eleven lettered nodes and seventeen connections. Four of the connections are asymmetric, shown with arrowheads denoting their direction. The other connections are undirected. One node, "Z", has no connections so cannot be reached from any of the others. The numbers indicate the distances between nodes.



Graph traversal algorithms, also known as *graph search algorithms*, follow the connections in graphs to

solve specific problems. A common problem is finding the shortest route between two nodes. This is used in network routing protocols and satellite navigation systems for road users.

Dijkstra's algorithm, named after *Edsger W Dijkstra*, is a graph traversal algorithm. Given a starting point within a graph, the algorithm determines the shortest route to every other connected node. Unconnected nodes, such as "Z" in the diagram, are assigned a distance of infinity.

Dijkstra's Algorithm

Dijkstra's algorithm includes five steps that calculate the distance of each node in a graph. The steps are:

1. Initialise the graph by setting the distance for every node to infinity, except for the starting node, which has a distance of zero. Mark every node in the graph as unprocessed.
2. Choose the current node. This is the unprocessed node with the smallest, non-infinite distance. If every node in the graph has been processed, or only nodes with an infinite distance remain unprocessed, the algorithm has completed. During the first iteration, the current node is the starting point.
3. Calculate the distance for all of the current node's unprocessed neighbours by adding the current distance to the length of the connection to the neighbour. For each neighbour, if the calculated distance is shorter than the neighbour's current distance, set the distance to the new value. For example, if the current node's distance from the starting point is 20 miles and the distance to a neighbour is 5 miles, the neighbour's potential distance is 25 miles. If the neighbour's distance is already less than 25 miles, it remains unchanged. If it is more than 25 miles, it's distance is overwritten and becomes 25 miles.
4. Mark the current node as processed.
5. Repeat the process from step 2.

.NET 3.5+

Graph traversal algorithms are used to process a graph of interconnected nodes, visiting each node by following a set behaviour. Dijkstra's algorithm is such an algorithm. It determines the shortest path from a start point to all other nodes in a graph.

In the remainder of the article we'll implement Dijkstra's algorithm using C#. This includes some LINQ standard query operators, which require .NET 3.5. If you are using an earlier version of the framework, you can use foreach loops and conditional processing to achieve the same results. The code also includes automatically implemented properties, which will need to be expanded.

Node Class

The first class that we need represents a node in a graph. This is an internal class as we only wish it to be

visible from within the project. If the algorithm and data structure classes are compiled into an assembly, this will prevent incorrect use of the data structures. The code for the Node class is shown below, with the description following:

```
internal class Node
{
IList<NodeConnection> _connections;

internal string Name { get; private set; }

internal double DistanceFromStart { get; set; }

internal IEnumerable<NodeConnection> Connections
{
get { return _connections; }
}

internal Node(string name)
{
Name = name;
_connections = new List<NodeConnection>();
}

internal void AddConnection(Node targetNode, double distance, bool twoWay)
{
if (targetNode == null) throw new ArgumentNullException("targetNode");
if (targetNode == this) throw new ArgumentException("Node may not connect to
itself.");
if (distance <= 0) throw new ArgumentException("Distance must be positive.");

_connections.Add(new NodeConnection(targetNode, distance));
if (twoWay) targetNode.AddConnection(this, distance, false);
}
}
```

The Node class includes three properties. The *Name* property allows us to provide a name for the node. This is a unique name within a graph. The *DistanceFromStart* property is populated by the algorithm. On completion of a calculation, this will hold the length of the shortest possible route from the starting node. The Connections property provides read-only access to the list of connections that can be followed from this node to its neighbours.

The constructor for the Node class requires that a name is provided when the object is initialised. This is

stored in the Name property. The constructor also initialises the Connections property as an empty list.

The final member, "AddConnection", is used to add a connection to a neighbour, creating a *NodeConnection* object, which is described in the next section. The target node, distance and a flag indicating if the connection is symmetric or asymmetric must be provided. The first three lines validate the provided parameters. Next the connection is added to the node. If the *twoWay* flag is set, a mirror connection is created from the target node to the current object.

NodeConnection Class

The NodeConnection class represents a one-way connection to a node. When you create a two-way connection, two NodeConnection objects are generated. The code for the class is shown below:

```
internal class NodeConnection
{
internal Node Target { get; private set; }
internal double Distance { get; private set; }

internal NodeConnection(Node target, double distance)
{
Target = target;
Distance = distance;
}
}
```

This class holds two properties for the target node and the distance between the pair of nodes. The source node is not included as NodeConnection objects are always used in the context of the source Node object.

Graph Class

The final data structure class is *Graph*. Objects of this type hold entire graphs containing nodes and connections.

```
public class Graph
{
internal IDictionary<string, Node> Nodes { get; private set; }

public Graph()
{
Nodes = new Dictionary<string, Node>();
}

public void AddNode(string name)
{
var node = new Node(name);
Nodes.Add(name, node);
}

public void AddConnection(string fromNode, string toNode, int distance, bool twoWay)
{
Nodes[fromNode].AddConnection(Nodes[toNode], distance, twoWay);
}
}
```

The Graph class includes a single internal member, which holds a dictionary of nodes. Nodes are added using the AddNode method. This creates a new node and adds it to the dictionary, using the name as the key. If you attempt to add two nodes with the same name, an exception will be thrown. The AddConnection method is used to create connections between neighbours.

.NET 3.5+

Graph traversal algorithms are used to process a graph of interconnected nodes, visiting each node by following a set behaviour. Dijkstra's algorithm is such an algorithm. It determines the shortest path from a start point to all other nodes in a graph.

DistanceCalculator Class

With the data structures in place we can now implement the algorithm. To begin, create a class named "DistanceCalculator" and add the method declaration shown below. The method has parameters to receive a graph and a starting node. It returns a dictionary containing the names of all of the nodes in the graph and their total distances from the starting point.

```
public class DistanceCalculator
{
public IDictionary<string, double> CalculateDistances(Graph graph, string
startingNode)
{
}
}
```

Before starting the calculations, we will check that the node name provided is actually within the graph and throw an exception if it is not.

```
if (!graph.Nodes.Any(n => n.Key == startingNode))
throw new ArgumentException("Starting node must be in graph.");
```

To complete the method we'll break the algorithm into three phases and create a further, private method for each. The first phase will initialise the graph as per step 1 in the algorithm description. The second method will traverse the graph and calculate the distances for each node. The final step will create and return the resultant dictionary of nodes and distances.

```
InitialiseGraph(graph, startingNode);
ProcessGraph(graph, startingNode);
return ExtractDistances(graph);
```

The initialisation step sets all of the node distances except the starting point to infinity. The starting node has a distance of zero.

```
private void InitialiseGraph(Graph graph, string startingNode)
{
foreach (Node node in graph.Nodes.Values)
node.DistanceFromStart = double.PositiveInfinity;
graph.Nodes[startingNode].DistanceFromStart = 0;
}
```

The *ProcessGraph* method contains the code that traverses the graph until all nodes are processed or only unreachable nodes remain. Rather than flagging each node as processed or unprocessed, a "queue" of nodes to be processed is generated. Initially this contains all of the nodes. As nodes are processed they are removed from the collection.

The while loop controls the processing of the node queue. In each iteration the next node to process is obtained by retrieving the first node from the queue when sorted by distance. The call to FirstOrDefault includes a filtering lambda expression that excludes nodes with an infinite distance. If there are no non-infinite distance nodes remaining, FirstOrDefault returns null and the loop exits. If a matching node is

present, this becomes the current node and is processed using the *ProcessNode* method.

```
private void ProcessGraph(Graph graph, string startingNode)
{
bool finished = false;
var queue = graph.Nodes.Values.ToList();
while (!finished)
{
Node nextNode = queue.OrderBy(n => n.DistanceFromStart).FirstOrDefault(n =>
!double.IsPositiveInfinity(n.DistanceFromStart));
if (nextNode != null)
{
ProcessNode(nextNode, queue);
queue.Remove(nextNode);
}
else
{
finished = true;
}
}
}
```

The next method is ProcessNode, which is responsible for calculating the distances for all neighbours of the current node. To do so it first obtains the list of connections from the current node to all unprocessed neighbouring nodes. We don't need processed neighbours as these already hold the correct shortest route distance.

The total distance to each of the neighbouring nodes is calculated by adding the connection distance to the DistanceFromStart value of the current node. If the calculated distance is shorter than the neighbours current DistanceFromStart, the property value is set to the new calculated value.

```
private void ProcessNode(Node node, List<Node> queue)
{
var connections = node.Connections.Where(c => queue.Contains(c.Target));
foreach (var connection in connections)
{
double distance = node.DistanceFromStart + connection.Distance;
if (distance < connection.Target.DistanceFromStart)
connection.Target.DistanceFromStart = distance;
}
}
```

The final method generates the results in a dictionary that can be used by the calling method without knowledge of the internal classes. It uses the ToDictionary standard query operator to generate a dictionary where the keys are the names of the nodes and the values are the distances.

```
private IDictionary<string, double> ExtractDistances(Graph graph)
{
return graph.Nodes.ToDictionary(n => n.Key, n => n.Value.DistanceFromStart);
}
```

Testing the Calculator

We can test the algorithm by creating a graph and traversing it. The code below creates the graph pictured at the start of this article. The distances from the "G" node are calculated and the results outputted. Try changing the distances or the starting point and viewing the results.

```
Graph graph = new Graph();

//Nodes
graph.AddNode("A");
graph.AddNode("B");
graph.AddNode("C");
graph.AddNode("D");
graph.AddNode("E");
graph.AddNode("F");
graph.AddNode("G");
graph.AddNode("H");
graph.AddNode("I");
graph.AddNode("J");
graph.AddNode("Z");

//Connections
graph.AddConnection("A", "B", 14, true);
graph.AddConnection("A", "C", 10, true);
graph.AddConnection("A", "D", 14, true);
graph.AddConnection("A", "E", 21, true);
graph.AddConnection("B", "C", 9, true);
graph.AddConnection("B", "E", 10, true);
graph.AddConnection("B", "F", 14, true);
graph.AddConnection("C", "D", 9, false);
graph.AddConnection("D", "G", 10, false);
graph.AddConnection("E", "H", 11, true);
graph.AddConnection("F", "C", 10, false);
```

```
graph.AddConnection("F", "H", 10, true);
graph.AddConnection("F", "I", 9, true);
graph.AddConnection("G", "F", 8, false);
graph.AddConnection("G", "I", 9, true);
graph.AddConnection("H", "J", 9, true);
graph.AddConnection("I", "J", 10, true);

var calculator = new DistanceCalculator();
var distances = calculator.CalculateDistances(graph, "G"); // Start from "G"

foreach (var d in distances)
{
Console.WriteLine("{0}, {1}", d.Key, d.Value);
}

/* OUTPUT

A, 28
B, 22
C, 18
D, 27
E, 29
F, 8
G, 0
H, 18
I, 9
J, 19
Z, Infinity

*/
```

Original URL:

http://www.blackwasp.co.uk/Dijkstra.aspx