The Layout Algorithm

At the heart of the force-directed algorithm are the methods that calculate the forces themselves:

The *repulsion* force is exerted by every node in the diagram, and each node is repelled (however slightly) by every other node in the diagram. This force is likened to the repulsion of charged particles, and so it is based on Coulomb's Law; **F = k(Q1Q2/r2)** – where **k** is a constant, **Q** is the charge and **r** is the distance between the particles. Since all of our nodes are equivalent, we can ignore the charge term. In other words, as the distance between the nodes increases, the repulsion force decreases quadratically. This calculation is performed in the ***CalcRepulsionForce()*** method:

```
private Vector CalcRepulsionForce(Node x, Node y) {
    int proximity = Math.Max(CalcDistance(x.Location, y.Location), 1);

    double force = -(REPULSION_CONSTANT / Math.Pow(proximity, 2));
    double angle = GetBearingAngle(x.Location, y.Location);

    return new Vector(force, angle);
}
```

The *attraction* force is exerted on each node by the nodes that are connected to it; isolated nodes are therefore unaffected by this force. The connectors between the nodes are regarded as spring-like, hence the force calculated using Hooke's Law; **F = −kx** – where **k** is a constant and **x** is the difference between the length of the spring and the distance between the nodes (i.e. the extension of the spring). In other words, as the distance between the nodes increases, the force pulling them back together increases proportionally (at least until the spring is fully relaxed, in which case the force stops applying). This calculation is performed in the ***CalcAttractionForce()*** method:

```
private Vector CalcAttractionForce(Node x, Node y, double springLength) {
    int proximity = Math.Max(CalcDistance(x.Location, y.Location), 1);

    double force = ATTRACTION_CONSTANT * Math.Max(proximity - springLength, 0);
    double angle = GetBearingAngle(x.Location, y.Location);

    return new Vector(force, angle);
}
```

The force-directed algorithm applies these forces to the nodes via an iterative simulation, as follows:

1. Randomise the initial coordinates of each **Node** – even a tiny variation will be sufficient.
2. Until the maximum number of iterations is exceeded (or the stop condition is satisfied):
    1. Initialise the *totalDisplacement* to zero.
    2. For each node:
        1. Initialise the *netForce* **Vector** to zero.
        2. Call ***CalcRepulsionForce()*** on every other node in the diagram, adding the result to *netForce*.
        3. Call ***CalcAttractionForce()*** on every node connected to the current node, adding the result to *netForce*.
        4. Add *netForce* to the node's velocity vector.
        5. Let the node's new position be set to the vector sum of its current position and its velocity.
    3. For each node:
        1. Calculate the displacement caused by moving the node to its new position, adding the result to *totalDisplacement*.
        2. Move the node to its new position.
    4. If *totalDisplacement* is less than a threshold value, trigger the stop condition.
3. Offset each node such that the diagram is centered around the origin (0,0). (During iteration, the diagram may creep.)

Usage Examples

Drawing a basic diagram to a bitmap

```
Diagram diagram = new Diagram();

// create some nodes
Node fruit = new SpotNode();
Node vegetables = new SpotNode();
Node apple = new SpotNode(Color.Green);
Node orange = new SpotNode(Color.Orange);
Node tomato = new SpotNode(Color.Red);
Node carrot = new SpotNode(Color.OrangeRed);

diagram.AddNode(fruit);
diagram.AddNode(vegetables);

// create connections between nodes
fruit.AddChild(apple);
fruit.AddChild(orange);
fruit.AddChild(tomato);
vegetables.AddChild(tomato);
vegetables.AddChild(carrot);

// run the force-directed algorithm
diagram.Arrange();

// draw to a Bitmap and save
Bitmap bitmap = new Bitmap(640, 480);
Graphics g = Graphics.FromImage(bitmap);
diagram.Draw(g, new Rectangle(Point.Empty, bitmap.Size));
bitmap.Save("demo.bmp");
```

Sample application

In the source code package (at the bottom of the article), I have included a simple Windows Forms application that demonstrates how to generate a random hierarchical diagram and paint it persistently on a **Form**. The layout is performed in a separate thread, and can optionally be displayed in a real-time, animated fashion.

Final Words

Force-directed algorithms are not perfect when it comes to arranging the nodes on a diagram; they are subject to conforming to local extremes rather than finding a 100%-optimal layout. This means that diagrams with a large number of nodes will typically have a higher occurrence of crossed lines. However, even when a less-optimal result is obtained, said result is often still visually striking. Running time can be a problem with particularly dense diagrams, but as already discussed, simpler diagrams produce a more aesthetically-pleasing result anyway. In most practical applications, a force-based approach provides a very good result for dynamically-generated diagrams.

With respect to my implementation, it is a very bare-bones approach, but it is also very scalable; for example, if appropriate node subclasses were written, it could be used to produce flow-charts, people-relationship diagrams, site maps for websites, brainstorming charts or even database diagrams (with some additional functionality). The ability to automatically generate such diagrams can open up a whole new world in terms of application usability, data analysis and the production of deliverables.

A force-directed algorithm will definitely drive my future diagramming/graphing projects.