* [Skip navigation](#content" \o "skip navigation)
* [Skip to search form](#q)

# March Column: Network Flow Algorithms

By [George T. Heineman](http://www.oreillynet.com/pub/au/3414)  
April 6, 2009 ([1ed](http://broadcast.oreilly.com/2009/01/january-column-graph-algorithm.html)) | Revised January 3, 2016 (2ed)

This is the fifth of a series of monthly columns in the blog associated with the [**Algorithms in a Nutshell**](http://oreilly.com/catalog/9780596516246/)**, 1ed** book, published October 2008 by [**O'Reilly Media, Inc.**](http://oreilly.com/)

[**In our last column**](http://broadcast.oreilly.com/2009/02/february-column-improving-algo.html) we described how to improve on the performance of the algorithm to solve Free Cell solitaire games.

In this column we present the material from **Chapter 8: Network Flow Algorithms**. Too often, this topic is omitted from undergraduate Computer Science courses on Algorithms. However, many disciplines have problems based on Network Flows, including Computer Science, Mathematics, Engineering and Industrial Engineering and Operations Research.

How would you solve the following ***Transportation Problem***?

|  |
| --- |
| You are the shipping manager in charge of two factories in Chicago and Washington, D.C. that can each produce 300 widgets daily. You must ensure that two customers in Houston and Boston each receive 300 widgets a day. You have several options for shipping, as shown below. For example, between Washington, D.C. and Houston, you may ship up to 280 widgets daily at $4.00 per widget, but the cost increases to $6.00 per widget if you ship from Washington, D.C. to Boston (although you can then send up to 350 widgets per day along that route). |
| |  |  |  | | --- | --- | --- | |  | Houston3 | Boston4 | | Chicago1 | 200 @ $7 | 200 @ $6 | | Wash., D.C.2 | 280 @ $4 | 350 @ $6 |   Your task is to find the cheapest shipping schedule to satisfy customer demand. |

To answer this question, we describe:

* The [**Maximum Flow problem**](#MF) over a Network Flow
* The [**Ford-Fulkerson**](#FF) technique for solving Maximum Flow
* [**Comparing Depth-first Ford-Fulkerson to Breadth-first Edmonds-Karp**](#comp)
* How a minor adjustment to the algorithm enables it to solve [**Maximum Flow, Minimum Cost problems**](#MFMC)
* How to reduce Minimum Cost Flow problems to a wide range of interesting problems

**But first a quick Plug...**

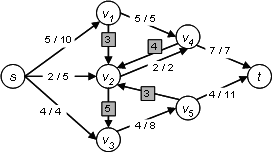
If you are interested in spending a day learning about many of the algorithms described in this book, consider joining me for a full-day seminar at MIT, sponsored by the Greater Boston ACM Chapter and the IEEE Boston Chapter. Entitled [**Algorithms Unleashed: A Practical Approach to Programming Algorithms**](http://www.oreillynet.com/pub/e/1249) this event will describe many of the algorithms in the book, and present the context and rationale behind the major design choices that lead to their efficiency gains.

**View March Code Samples**

The code samples are found on the github repository (<https://github.com/heineman/algorithms-nutshell-2ed.git>) in the Blogs/ project.

**Maximum Flow problem over a Network**

A common problem that arises in multiple disciplines is called the Maximum Flow problem. Imagine you are in charge of a heating system which consists of a series of pipes between different locations. Each pipe has a maximum *capacity* which it can deliver from point A to point B. For example, assume you have the following arrangement:

  
**Fig. 1** Example network flow with sample flows and capacities.

One way to interpret this image is to consider vertex "s" to be the source of water entering the heating system and the vertex "t" represents the sink (or target) from which all water exits the system. Each pipe has a fixed maximum capacity and you can adjust its flow to limit or increase the amount of water flowing between the two locations (from nothing up to and including this maximum capacity). For example, from the source location "s", a total of 10 units of water can flow to location v1. In the configuration above, there are currently 5 units (out of a possible 10) flowing from s to v1. Similarly, there are 2 units (out of a possible 5) flowing from s to v2. The edges marked with gray boxes show the potential capacity where no flow as of yet exists (i.e., the pipe is turned off). For example, you could allow up to 3 units to flow from v1 to v2.

The network can be arbitrarily complex; for example, note that there is a pipe connecting v2 to v4 (and 2 units out of a possible 2 are flowing in that direction) but there is an additional pipe that connects v4 back to v2 (and while it currently has no water flowing through it, up to 4 units could flow).

In a network flow, there are certain global properties that must be maintained:

* *Capacity Constraint* The flow f(u,v) through an edge cannot be negative and cannot exceed the capacity of the edge c(u,v), 0 ≤ f(u,v) ≤ c(u,v). If an edge (u,v) doesn't exist in the network, then c(u,v) = 0.
* *Flow Conservation* Aside from the source vertex "s" and the sink vertex "t", each vertex must satisfy the property that the sum of f(v,u) for all edges (v,u) that flow into vertex u must equal the sum of f(u, w) for all edges (u,w) that flow out of u.

Given the configuration above, you can verify that 11 units of water flow out of vertex "s" and that 11 units of water flow into vertex "t". Is it possible to increase the flow of water through the system? It certainly looks possible, since there is a potential of 10+5+4 = 19 units that can flow out of vertex "s", while vertex "t" can handle up to 11+7 = 18 units.

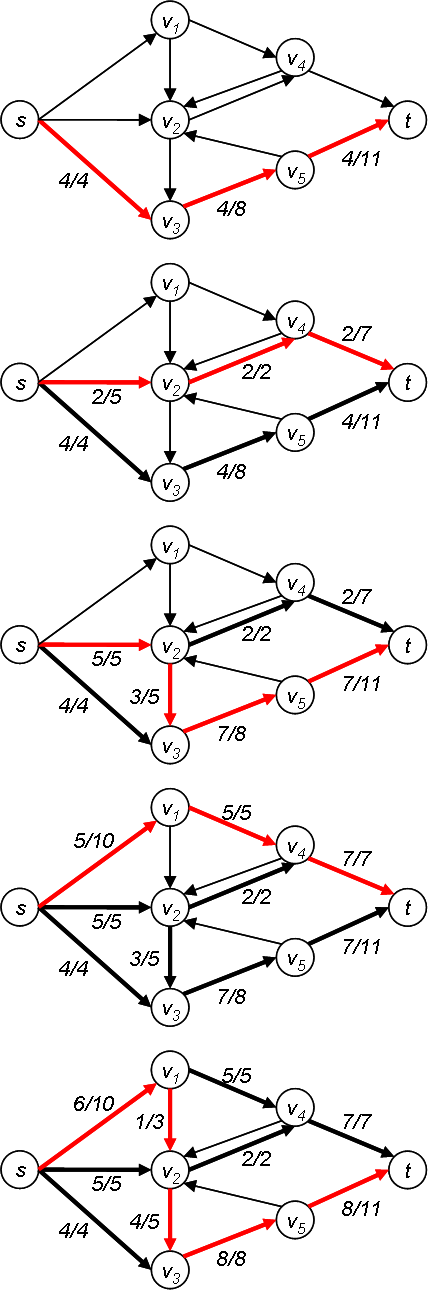
How many possible configurations are there? It is assumed that all flow units are integral (i.e., there is no way to ship 1/2 a unit of water). The total number of configurations is the multiplication of (capacity+1) for each pipes. For example, if a pipe has capacity of 5, then there are six possible flow configurations: {0, 1, 2, 3, 4, 5}. Thus the total product = 11\*6\*5\*4\*6\*6\*5\*3\*4\*9\*8\*12 = 2,463,436,800; not all of these are valid. A brute force approach must try each one. Assuming you can compute and process 12,000,000 configurations per second (your mileage may vary!), you could locate one which gives you the maximum flow in just under 4 minutes. For the record, the maximum flow for this particular network is actually 15 units; we will see in the next section how to compute this value using a more efficient algorithm.

The **BruteForce** class provided with the source code for this month's blog contains the implementation of such a brute force approach. Surely there must be a better way, since the brute force method doesn't take any advantage of the structure of the network.

**The** [**Ford-Fulkerson**](#FF) **technique for solving Maximum Flow**

The Ford-Fulkerson algorithm is able to compute the Maximum Flow by repeatedly asking the question "can I add just a little bit more to an existing Network?" More precisely, it searches for an *augmenting path* through the network from source "s" to target "t" to which more flow can be added. Ford-Fulkerson repeats this process until no such augmenting path can be found. An important result known as the Max-flow Min-Cut theorem guarantees that, with non-negative flows and capacities, Ford-Fulkerson always terminates and identifies the maximum flow in a network.

Given the network from **Fig. 1**, Ford-Fulkerson requires five passes to converge on the maximum flow, as shown in **Fig. 2** below. The augmenting paths are marked in red and edges are labeled "f/c" to represent the non-zero flow and the available capacity (edges with no flow have no label).

  
**Fig. 2** Ford-Fulkerson solution for sample Network in Fig. 1.

In the example in **Fig. 2**, the computed augmenting paths are computed strictly by adding new flow units incrementally. The Ford-Fulkerson algorithm actually handles the more complex case where it can discover how to increase the overall flow through the network by occasionally reducing flow along one edge, and re-routing that flow elsewhere. Figure 8-3 in the book contains an example that shows how this situation arises.

The key insight of Ford-Fulkerson is that one can use any search method to locate an augmenting path, then increase the flow by the computed amount along that path. The core of the algorithm is contained in the code fragment below.

**public boolean** compute () {

  boolean augmented = false;

**while** (searchMethod.findAugmentingPath(network.vertices)) {

    processPath(network.vertices);

    augmented = **true**;

  }

**return** augmented;

}

**Example 1**: algs.model.network.FordFulkerson code fragment.

The original Ford-Fulkerson algorithm used a Depth-first search over the network to locate the augmenting paths; when a Breadth-first search is used, the algorithm is known as Edmonds-Karp. The code below contains the Depth-first search (DFS) implementation over a network represented using an adjacency matrix to represent the network:

**public boolean** findAugmentingPath (VertexInfo []vertices) {

vertices[sourceIndex] = **new** VertexInfo (-1);

Stack path = **new** Stack();

path.push (sourceIndex);

**while** (!path.isEmpty()) {

**int** u = path.pop();

// find all unvisited vertices connected to u by edges.

**for** (**int** v = 0; v < numVertices; v++) {

EdgeInfo inf;

// only visit if not self and not already visited

**if** (v == u) **continue**;

**if** (vertices[v] != **null**) **continue**;

**if** (((inf = info[u][v]) != **null**)&&(inf.capacity > inf.flow)) {

// increase forward flow, if edge exists and capacity

// still allows extra units to flow

vertices[v] = **new** VertexInfo (u, FORWARD);

} **else if** (((inf = info[v][u]) != **null**)) {

**if** (inf.flow > 0) {

// decrease reverse flow, if reverse edge exists and

// units are flowing along the edge

vertices[v] = **new** VertexInfo (u, BACKWARD);

}

}

**if** (vertices[v] != **null**) {

**if** (v == sinkIndex) {

// we now have a path from source to sink.

**return true**;

} **else** {

// continue looking for a path with this vertex

path.push(v);

}

}

}

}

// nothing

**return false**;

}

**Example 2**: algs.model.network.DFS\_SearchArray code fragment.

As you might expect from a Depth-first strategy, a stack is used to store the progress of the potential augmenting path. The code repository contains implementations **DFS\_SearchArray**, **BFS\_SearchArray**, **DFS\_SearchList** and **BFS\_SearchList**.

**Comparing Depth-first Ford-Fulkerson to Breadth-first Edmonds-Karp**

Does it really matter which variation you use? And does it matter whether the underlying network representation uses adjacency lists or adjacency matrices? Naturally, the answer depends upon the structure of the graph; however, it also depends upon the potential maximum flow value itself. As it turns out, the average-case performance of Ford-Fulkerson is O(E\*mf) where *mf* is the actual maximum flow; the reason is that at each iteration, Ford-Fulkerson might only increment the total flow through the network by a single unit, which means it may require lots of iterations. By contrast, Edmonds-Karp experiences an average-case performance of O(V\*E2).

In practice, either approach provides an efficient solution; only truly contrived networks will cause a problem.

**Maximum Flow, Minimum Cost problems over a Network**

Let us now return to our original [***Transportation Problem***](#transportation) as stated at the beginning of this column. Consider if both factories ships 150 widgets each to Houston and Boston. In this schedule, both customers each receive 300 widgets. What is the cost of this schedule?

Cost = 150\*7 + 150\*6 + 150\*4 + 150\*6 = $3,450

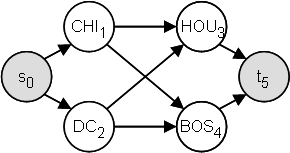
Is it possible to construct a cheaper schedule? Somehow we must determine the best cost from a staggering number of options. How many? Well, if we assume that we can ship any integral number of widgets between cities, then the total number of possible shipping configurations is equal to:

NS = 201 \* 201 \* 281\* 351 = 3,984,791,031 (or just under 4 billion configurations).

If this problem involved more factories or customers, the sheer number of potential configurations would become unimaginable.

For many computer programmers, this type of problem is completely unfamiliar. It doesn't look like a **sorting** problem, or even a **search**. We can't even fall back on a tried-and-true **Brute-Force** approach as we did earlier because there are simply too many configurations to check.

First, we convert the problem to create a network with a unique source vertex and a unique target vertex. This conversion is a typical exercise in computer science, where you convert the input of one problem to be used by the solution of another problem. In our case, we need only add two additional vertices (s0 and t5), edges from s0 to each factory (with capacity equal to the production rate of each factory), and edges from the customers to the target t5 (with capacity equal to the demand of each customer). The converted network is shown in **Fig. 3**

  
**Fig. 3** Converted Network

Now running Ford-Fulkerson on the network in Fig. 3 results in a computed maximum flow of 600 units (as we would expect, with 300 units flowing each from the two factories to the two customers). The specific configuration computed (as shown in Figure 8-7 in the book) costs $3,600 to ship the 600 units. However, we already found a shipping schedule above that cost only $3,450. Can we do better? The short answer is yes (as we shall see), but the challenge is being able to identify an algorithm that will produce the desired result.

You may recognize that both Depth-first (Ford-Fulkerson) and Breadth-first (Edmonds-Karp) searches are "blind" to the actual cost, since they strictly follow the structure of the network as the search ensues. We have already seen greedy algorithms (such as Prim's Algorithm for building a Minimum Spanning Tree in Chapter 6) that iteratively operate by extending a search by choosing the least costly direction to pursue; perhaps such an approach will work here.

We can use a **priority queue** to associate with each vertex, v, a priority which reflects the cost of shipping one additional unit from the source to v. As the search progresses, the vertices in the priority queue define the active "search horizon" being investigated. Each vertex, v, in the priority queue has an associated cost that reflects the least expensive cost for shipping a single unit from the source vertex to that vertex "v" along some sequence of edges. To locate the least expensive augmenting path, the search proceeds by extending the search horizon (i.e., the vertices in the priority queue) from the vertex with least cost in the priority queue. If this expansion encounter the sink vertex, then the method locates the least cost augmenting path for the current network.

**public boolean** findAugmentingPath (VertexInfo[] vertices) {

Arrays.fill(vertices, **null**); // reset for iteration

// Construct queue using BinaryHeap. The inqueue[] array avoids

// an O(n) search to determine if an element is in the queue.

**int** n = vertices.length;

BinaryHeap pq = **new** BinaryHeap(n);

**boolean** inqueue[] = **new boolean** [n];

// initialize dist[] array. Use INT\_MAX when edge doesn't exist.

**for** (**int** u = 0; u < n; u++) {

**if** (u == sourceIndex) {

dist[u] = 0;

pq.insert(sourceIndex, 0);

inqueue[u] = **true**;

} **else** {

dist[u] = Integer.MAX\_VALUE;

}

}

**while** (!pq.isEmpty()) {

**int** u = pq.smallestID();

inqueue[u] = **false**;

// When reach sinkIndex we are done.

**if** (u == sinkIndex) { **break**; }

for (int v = 0; v < n; v++) {

**if** (v == sourceIndex || v == u) **continue**;

// forward edge with remaining capacity if cost is better.

EdgeInfo cei = info[u][v];

**if** (cei != null && cei.flow < cei.capacity) {

**int** newDist = dist[u] + cei.cost;

**if** (0 <= newDist && newDist < dist[v]) {

vertices[v] = **new** VertexInfo (u, Search.FORWARD);

dist[v] = newDist;

**if** (inqueue[v]) {

pq.decreaseKey(v, newDist);

} **else** {

pq.insert(v, newDist);

inqueue[v] = **true**;

}

}

}

// backward edge with at least some flow if cost is better

cei = info[v][u];

**if** (cei != **null** && cei.flow > 0) {

**int** newDist = dist[u] - cei.cost;

**if** (0 <= newDist && newDist < dist[v]) {

vertices[v] = **new** VertexInfo (u, Search.BACKWARD);

dist[v] = newDist;

**if** (inqueue[v]) {

pq.decreaseKey(v, newDist);

} **else** {

pq.insert(v, newDist);

inqueue[v] = **true**;

}

}

}

}

}

**return** dist[sinkIndex] != Integer.MAX\_VALUE;

}

**Example 3**: algs.model.network.ShortestPathArray code fragment.

You can launch this version of Ford-Fulkerson by the following code snippet (which can be found in the algs.chapter8.figure7 package of the Figures section of the ADK) which processes the given flow network:

FlowNetworkArray network; // assume properly constructed...

ff = new FordFulkerson (network, new ShortestPathArray(network));

ff.compute();

**Example 4**: Code snippet for computing Max Flow Min Cost

As you can verify upon executing the code, the least expensive shipping schedule is $3,300 which asks for:

* Chicago to send 100 units to Houston (@ $7)
* Chicago to send 200 units to Boston (@ $6)
* DC to send 200 units to Houston (@ $4)
* DC to send 100 units to Boston (@ $6)

This schedule costs $3,300 = 100\*7 + 200\*6 + 200\*4 + 100\*6.

**Power of Linear Programming**

The transportation problem presented in this column can also be solved using the more powerful technique known as Linear Programming. The **Simplex Algorithm**, designed by Danzig in 1947, has been cited as one of the top ten algorithms of the twentieth century, according to the editors of [**Computing in Science and Engineering**](http://www2.computer.org/portal/web/csdl/doi/10.1109/MCISE.2000.814652). Fortunately there is no need for you to implement this algorithm since it is provided as part of most mathematical software, such as [**Maple**](http://www.maplesoft.com/) and [**Mathematica**](http://www.wolfram.com/). For example, to use Maple to solve the transportation problem earlier, the following sequence of commands will do the job:

with(simplex);

Constraints := [

# conservation of units at each node

e13+e14 = 300, # CHI

e23+e24 = 300, # DC

e13+e23 = 300, # HOU

e14+e24 = 300, # BOS

# maximum flow on individual edges

0 <= e13, e13 <= 200,

0 <= e14, e14 <= 200,

0 <= e23, e23 <= 280,

0 <= e24, e24 <= 350

];

Cost := 7\*e13 + 6\*e14 + 4\*e23 + 6\*e24;

# Invoke linear programming to solve problem

minimize (Cost, Constraints, NONNEGATIVE);

**Example 5**: sample Maple commands to compute same solution

If you have **Maple** you can type the above commands, and the resulting computation yields {e13 = 100, e24 = 100, e23 = 200, e14 = 200} which is the same result as computed by Example 3.

**Validating the algorithms**

It is a challenge to properly test the execution of the algorithms in this chapter. You can easily test the behavior of some algorithms, such as **Sorting** and **Searching**, because the result of the algorithm can be easily verified (i.e., is the result sorted, or can a particular item be found when searched). However, testing network flow algorithms is challenging, because the algorithms must find a guaranteed "maximum" solution, or one that exhibits the maximum flow with the least amount of cost. To properly evaluate the algorithms in this chapter for our book, we scoured the available literature for examples; while this will not prove that the code is correct, we became more confident with our implementations when they were able to reproduce the solutions published in text books and academic papers.

Within the Algorithm Development Kit (ADK) that can be downloaded from the [**ADK Release**](http://examples.oreilly.com/9780596516246/Releases/) directory, you will find two directories, Tests and PerformanceTests, which contain the results of our testing. There are numerous articles from *Operations Research*, written during the 1950s and 1960s, which contain the original work that led to the development of Ford-Fulkerson and other like-minded algorithms (here is a [**1962 paper on the Transshipment problem**](http://www.jstor.org/pss/168052)).

These network flow algorithms solve economic problems and military logistics problems, and contributed to the early adoption of Information Technology by big business. You might recall that the first general purpose computer,  [**ENIAC**](http://en.wikipedia.org/wiki/Eniac), was designed and built to calculate artillery firing tables for the U.S. Army's Ballistic Research Laboratory. The network flow algorithms developed in the 1950s, and the widespread adoption of the [**Simplex Algorithm**](http://en.wikipedia.org/wiki/Simplex%20algorithm), helped drive the growth and productivity of worldwide commerce.

**Next Column**

In next Month's April column, we will turn our attention to investigate algorithms from **Chapter 9: Computational Geometry**. Until next time, we hope you take the opportunity to investigate the numerous algorithms in the Algorithms in a Nutshell book as well as to explore the examples provided in the ADK.

[**Algorithms in a Nutshell**](http://oreilly.com/catalog/9780596516246/)  
George T. Heineman, Gary Pollice, Stanley Selkow