# Network Flow

Senior Computer Team 2002

Presented by Tim Abbott

## 1 Introduction

Network flow is a difficult algorithm that often shows up on USACO contests, and TJ tends to do well on Network Flow problems (lectures help we think). This lecture is intended to ensure that you will know network flow, as even ”blatant” network flow programs become very valuable because most competitors do not know the algorithm. Learn this algorithm and you **will** benefit on at least one USACO contest this year.

Imagine that you have a system of broadband pipes throughout your city , each of which have a given capacity per second, and since you like fast internet connections, you would like to get as much bandwidth from the network to your house as possible. For this **network** of pipes, what is the maximum information **flow** per second that can reach your house?

In case you didn't guess from the subtle hints and the great big title at the top of this page, this is the classic network flow problem. Given a graph $G(V, E)$, of which two different vertices are the *source* and *sink*, find the maximum flow between them.

## 2 The Basic Algorithm

There is a theorem which we won't prove here because we don't know the proof. But there is another theorem whose proof we won't show that helps solve the network flow problem, and it is the basis behind the Ford-Fulkerson algorithm. This algorithm works on the principle that if you can find a path from source to sink which can still carry some amount of flow, send that flow through the pipes. If there are no more paths that can carry flow, you have already sent as much flow as possible through the network.

So we could find a path, send one unit of flow through it, and repeat. But wait! This is at best order $O(EF)$, where $F$ is the final flow, and this could be ridiculously large. So basic Ford-Fulkerson is not used. Instead, each time we find a path, we send as much flow as possible.

There are two ways to choose a path. One is to find the shortest path from source to sink, and the other is to find the path that will allow the most flow. Of course, when finding these paths you should ignore saturated edges, since no more flow can get through them anyway. Both of these algorithms are implemented in a way very similar to Dijkstra's algorithm. In general, it is preferable to use the latter algorithm because your goal is to send as much flow as possible to the sink.

We will provide neither code nor complete pseudocode for this algorithm. It is long, so I suggest you try coding it at home. If you wish to try it, try the Drainage Ditches problem on our grader. Here is pseudocode from USACO, made brief so that you have to think a little, and to save paper.

```
 5   while (True)
 6 # find path with highest capacity from
# source to sink
 7 # uses a modified djikstra's algorithm
 8     for all nodes i
 9       prevnode(i) = nil
10       flow(i) = 0
11       visited(i) = False
12     flow(source) = infinity
13     while (True)
14       maxflow = 0
15       maxloc = nil
16       # find the unvisited node with
# the highest capacity to it
21       if (maxloc = nil)
22         break inner while loop
23       if (maxloc = sink)
24         break inner while loop
24a      visited(maxloc) = true
25       # update its neighbors values of flow, prev
30     if (maxloc = nil)
# no path
31       break outer while loop
33     totalflow = totalflow + flow(sink)
# add that flow to the network,
# update capacity appropriately
35     curnode = sink
# for each arc, prevnode(curnode),
# curnode on path: (reversing flow)
36     while (curnode != source)
38       nextnode = prevnode(curnode)
39       capacity(nextnode,curnode) = capacity(nextnode,curnode) - pathcapacity
41       capacity(curnode,nextnode) = capacity(curnode,nextnode) + pathcapacity
43       curnode = nextnode
```

That's pretty much all there is to the basic network flow algorithm. There are many uses for this algorithm. Some of them are detailed below. Others are not, because those uses aren't needed on USACO problems.

## 3 Bipartite Matching

Let's say that you have a group of cows and a set of gifts to give them. But these cows are picky; they each have their own wish list and are only happy if you give them a gift they want. If each cow can only receive one gift, how many cows will be happy if you give gifts optimally?

In this problem we have to assign one set of objects (gifts) to another (cows). Assuming that the cows themselves cannot be gifts, we can divide the graph into two parts, or as they say in Latin, "bi parts" to match. Hence the name of the type of problem, "bipartite matching."

But network flow only deals with flow from one vertex to another, not a whole set to another whole set! We must add two more vertices: a Supersource connected only to the cows (but all the cows), and a Supersink connected only to the gifts (but all the gifts). What are the edge capacities? Well, each cow only gets one gift, so the capacity from source to cow MUST be 1 for each cow. Then, we can only give as many of a specific gift as there are copies to give. So the capacity from any gift to the sink is the number of that gift we have to give. What about the capacity from the cows to the gifts? Simple answer: it doesn't matter, as long as it's at least 1. (Prove it!)

There are two other [often at least one is necessary] optimizations we can make. First, finding a path to augment is $O(V^2)$. That is fairly slow. Before you start doing the entire flow algorithm, just greedily run through the list of possible source$-$ >cow$-$ >gift$-$ >sink combinations, and if you can send flow through, do it. Remember to reverse flow even on this part. And keep in mind you still need to do the regular algorithm, since this does not catch paths of greater lengths which you would get by reversing some flow. However if we look at Tim's runtimes for a camp bipartite matching problem (”poker”)using this greedy method and not doing so, run on the elements (about 3x as fast as Rob's grader):

| Case | N | Using | Not Using |
|---|---|---|---|
| 1 | 2 | 0.00 | 0.00 |
| 6 | 80 | 0.00 | 0.01 |
| 7 | 150 | 0.00 | 0.03 |
| 8 | 300 | 0.00 | 0.29 |
| 9 | 500 | 0.01 | 1.40 |

we see that this optimization is a Good Idea (TM). Since often the test data are much harder than these, you should use this greedy trick to save processing time. In general the network flow algorithm runs in approximately $O(V^3)$ while greedy runs in $O(V^2)$, so since $V$ generally goes up to 500 or 1000, we're saving at least 2.5 orders of magnitude (the greedy algorithm does the vast majority of the work the netflow would do).

The other optimization is that we can restrict the picking a path section of the algorithm so that the program ”knows” when it is using cows and does not try to match them to other cows and similary when it is using gifts. This optimization often provides a factor of 2 in runtimes.

## 4 Min cut

A malicious cow, coincidentally named Time, has poisoned the town water supply! Given the pipes carrying water around town and the cost to shut down each, find the least cost you need to spent to protect yourself from Time-corrupted water. Break ties by the one using the least number of pipes.

Solution: This one is actually fairly hard. Suppose there are N edges in the graph. Multiply the cost of each edge by $(N + 1)$ and add one. Compute the maximum flow using the standard method, and to get the cost you use floor($totflow/(N+1)$). To find the number of edges you use $totflow\%(N + 1)$ – but be careful!. We will be using doubles here (if cost can be big), except in the case that the costs are all equal (in which case the multiplication by $(N+1)$ was unecessary). Question: Why does this work? (this is hard, but its worth the thought)

To find the edges themselves, run a floodfill (you should remember this) from the source to mark all nodes that are still connected to the source. All edges that connect marked nodes to unmarked nodes are members of the optimal cut.

## 5 Challenge Problems

Naturally there is a great deal more that you can do with network flow. But you won't see other uses in a USACO contest. So here are some problems you may encounter, but beware! One of them can be solved with network flow but has a better solution. Can you pick it out?

1. **Green Bay Packer** [Burch, 2001]
   Farmer John is about to move, and he must put all of his belongings into boxes. He has $N$ $(1 \leq N \leq 1000)$ different tupes of objects that must each be put into their own box along with $S$ $(1 \leq S \leq 1000)$ boxes of different sizes. The box sizes are numbered in increasing sizes, starting at 1. Each object will fit only into a certain range of box sizes. What is the greatest number of objects that he can pack?

2. **Sun Protection** [Stroe]
   The calves are in cow-heaven: they have another way to make money. The beach-umbrella shop wants them to cover all the cows on the beach with umbrellas and will pay them handsomely to do so. These are interesting umbrellas - they cover a spot of beach two units by one unit. The vendor has zillions of them. The beach has been broken into a $W$x$H$ tiled grid $(1 \leq W, H \leq 250)$. Each umbrella can be placed on a pair of adjacent (vertical or horizontal, not diagonal) squares upon which cows lie. An umbrella can be placed only over two squares occupied by cows and cannot overlap with any other umbrella. Find the largest number of umbrellas that can be placed on the beach.

3. **Homework:** Solve Drainage Ditches. Hint: What happens if multiple pipes do the same thing?