Network Flow Algorithms

Prerequisite

- Shortest Path

The Problem

Given: A direct connected graph with integer weighted arcs, along with a source node and a sink node.

Each arc weight corresponds to the ``capacity'' of that arc. A flow through the graph is constructed by assigning an integer amount of ``flow'' to send through each edge such that:

- The flow through each arc is no greater than the arc's capacity.
- For each node other than the source and sink, total flow in is the same as total flow out.

Maximize the total of the weights of the out-arcs of the source minus the weights of the in-arcs (or the total of the weights of the in-arcs of the sink minus the weights of the out-arcs).

Example

Given: The layout of a collection of water pipes, along with the capacity of each pipe. Water in these pipes must flow downhill, so within each pipe, water can only flow in one direction.

Calculate the amount of water that can flow from a given start (the water-purification plant) to a given end (your farm).

The Algorithm

The algorithm (greedily) builds the network flow by iteratively adding flow from the source to the sink.

Start with every arc having weight equal to the beginning weight (The arc weights will correspond to the amount of capacity still unused in that arc).

Given the current graph, find a path from the source to the sink across arcs that all have non-zero weight in the current graph. Calculate the maximum flow across this path, call it PathCap.

For each arc along the path, reduce the capacity of that arc by PathCap. In addition, add the reverse arc (the arc between the same two nodes, but in the opposite direction) with capacity equal to PathCap (if the reverse arc already exists, just increase its capacity).

Continue to add paths until none exist.

This is guaranteed to terminate because you add at least one unit of flow each time (since the weights are always integers), and the flow is strictly monotonically increasing. The use of an added reverse arc is equivalent to reducing the flow along that path.

If you are interested in a more detailed analysis of this algorithm, consult Sedgewick.

Here is pseudocode for the algorithm:

```
 1   if (source = sink)
 2     totalflow = Infinity
 3     DONE

 4   totalflow = 0

 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
17       for all nodes i
18         if (flow(i) > maxflow AND
                         not visited(i))
19           maxflow = flow(i)
20           maxloc = i
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
26       for all neighbors i of maxloc
27         if (flow(i) < min(maxflow,
                     capacity(maxloc,i)))
28           prevnode(i) = maxloc
29           flow(i) = min(maxflow,
                    capacity(maxloc,i))

30     if (maxloc = nil)         # no path
31       break outer while loop

32     pathcapacity = flow(sink)
33     totalflow = totalflow + pathcapacity

   # add that flow to the network,
   # update capacity appropriately
35     curnode = sink
         # for each arc, prevnode(curnode),
         # curnode on path:
36     while (curnode != source)
38       nextnode = prevnode(curnode)
39       capacity(nextnode,curnode) =
           capacity(nextnode,curnode) -
40                          pathcapacity
41       capacity(curnode,nextnode) =
           capacity(curnode,nextnode) +
42                          pathcapacity
43       curnode = nextnode
```

Running time of this formulation is O(F M), where F is the maximum flow and M is the number of arcs. You will generally perform much better, as the algorithm adds as much flow as possible every time.

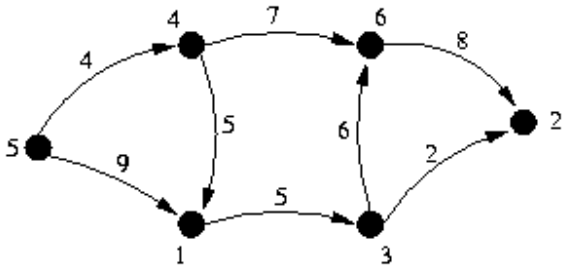
To determine the flow across each arc, compare the starting capacity with the final capacity. If the final capacity is less, the difference is the amount of flow traversing that

arc.

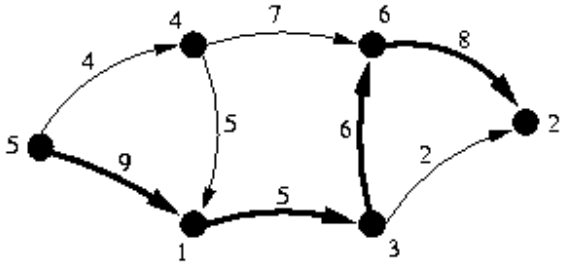
This algorithm may create `eddies,' where there is a loop which does not contribute to the flow itself.

Execution Example

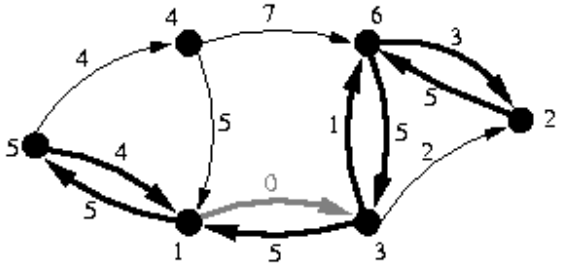
Consider the following network, where the source is node 5, and the sink is node 2.



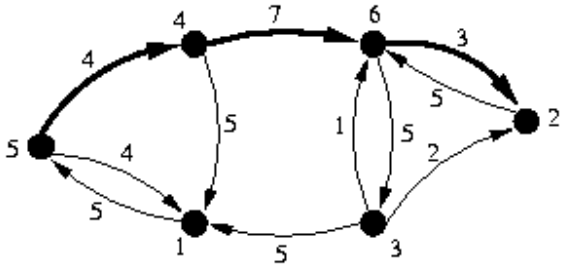
The path with the highest capacity is {5,1,3,6,2}.



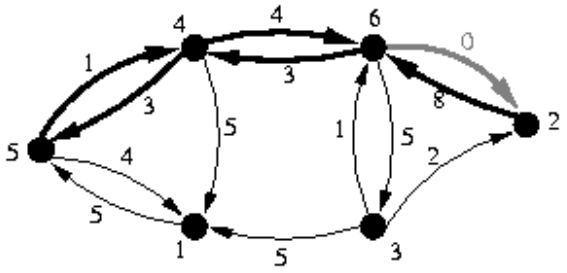
The bottleneck arc on this path is 1->3, which has a capacity of 5. Thus, reduce all arcs on the path by 5, and add 5 to the capacity of the reverse arcs (creating the arcs, if necessary). This gives the following graph:



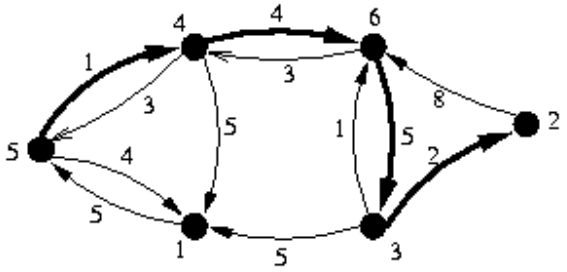
In the new graph, the path with highest capacity is {5,4,6,2}.



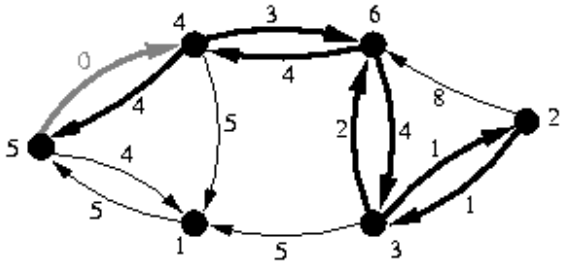
The capacity of this path is 3, so once again, reduce the forward arcs by 3, and increase the reverse arcs by 3.



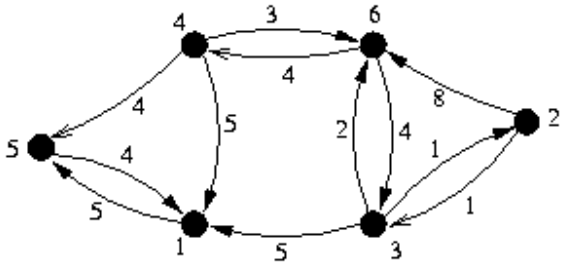
Now the network's maximum capacity path is {5,4,6,3,2}



This flow has only a capacity of 1, as the arc from 5 to 4 has capacity 1. Once again, update the forward and backwards arcs appropriately.



The resulting graph has no paths from the source to the sink. The only nodes reachable from the source node 5 are node 5 itself and node 1.



The algorithm added three flows, the first with capacity 5, the second with capacity 3, and the last with capacity 1. Thus, the maximum flow through the network from node 5 to node 2 is 9.

Extensions

Network flow problems are very extensible, mostly by playing with the graph.

To extend to the case of undirected graphs, simple expand the edge as two arcs in opposite directions.

If you want to limit the amount of traffic through any node, split each node into two nodes, an in-node and and out-node. Put all the in-arcs into the in-node, and all of the

out-arcs out of the out-node and place an arc from the in-node to the out-node with capacity equal to the capacity of the node.

If you have multiple sources and sinks, create a `virtual source' and `virtual sink' with arcs from the virtual source to each of the sources and arcs from each of the sinks to the virtual sink. Make each of the added arcs have infinite capacity.

If you have arcs with real-valued weights, then this algorithm is no longer guaranteed to terminate, although it will asymptotically approach the maximum.

Alternative Problems

Network flow can also be used to solve other types of problems that aren't so obvious

Maximum Matching

Given a two sets of objects (call them A and B), where you want to `match' as many individual A objects with individual B objects as possible, subject to the constraint that only certain pairs are possible (object A1 can be matched with object B3, but not object B1 or B2). This is called the `maximum matching' problem.

To reformulate this as network flow, create a source and add an arc with capacity 1 from this source to each A object. Create a sink with an arc from each B object to it with capacity 1. In addition, if object Ai and Bk may be matched together, add an arc from Ai to Bk with capacity 1. Now run the algorithm and determine which arcs between A objects and B objects are used.

Minimum Cut

Given a weight undirected graph, what is the set of edges with minimum total weight such that it separates two given nodes.

The minimum total weight is exactly the flow between those two nodes.

To determine the path, try removing each edge in increasing weight order, and seeing if it reduces the network flow (if it does, it should reduce the flow by the capacity of that edge. The first one which does is a member of the minimum cut, iterate on the graph without that edge.

This can be extended to node cuts by the same trick as nodes with limited capacity. Directed graphs work using the same trick. However, it can not solve the problem of finding a so-called `best match,' where each pairing has a `goodness' value, and you want to create the matching which has the highest total `goodness.'

Example Problems

If the problems talks about maximizing the movement or flow of something from one location to another, it is almost assuredly maximum flow. If it talks about trying to separate two items minimally, it is probably minimum cut. If it talks about maximizes the pairing of any sort of thing (person, object, money, whatever), it is probably maximum matching.

Virus Flow

You have a computer network, with individual machines connected together by wires. Data may flow either direction on the wire. Unfortunately, a machine on your network has caught a virus, so you need to separate this machine from your central server to stop the spread of this virus. Given the cost of shutting down the network connection between each pair of machines, calculate the minimum amount of money which must be spent to separate the contaminated machine from your server.

This is exactly the min cut problem.

Lumberjack Scheduling

Different types of trees require different techniques to be employed by lumberjacks for them to harvest the tree properly. Regardless of the tree or lumberjack, harvest a tree requires 30 minutes. Given a collection of lumberjacks, and the types of trees that each one is able to correctly cut down, and a collection of trees, calculate the maximum number of trees which may be cut down in the next half hour.

Each lumberjack can be paired with each tree of a type that he/she is able to properly harvest. Thus, the problem can be solved using the maximum matching algorithm.

Telecowmunication (USACO Championship 1996)

Given a group of computers in the field, along with the wires running between the computers, what is the minimum number of machines which may crash before two given machines are the network are unable to communicate? Assume that the two given machines will not crash.

This is equivalent to the minimum node cut problem. The two given machines can be arbitrarily labeled the source and sink. The wires are bidirectional. Split each node into an in-node and an out-node, so that we limit the flow through any given machine to 1. Now, the maximum flow across this network is equivalent to the minimum node cut.

To actually determine the cut, iterative remove the nodes until you find one which lowers the capacity of the network.

Science Fair Judging

A science fair has N categories, and M judges. Each judge is willing to judge some subset of the categories, and each category needs some number of judges. Each judge is only able to judge one category at a given science fair. How many judges can you assign subject to these constraints?

This is very similar to the maximum matching problem, except that each category can handle possibly more than one judge. The easiest way to do this is to increase the capacity of the arcs from categories to the sink to be the number of judges required.

Oil Pipe Planning

Given the layout (the capacity of each pipe, and how the pipes are connected together) of the pipelines in Alaska, and the location of each of the intersections, you wish to increase the maximum flow between Juneau and Fairbanks, but you have enough money to only add one pipe of capacity X. Moreover, the pipe can only be 10 miles long.

Between which two intersections should this pipe be added to increase the flow the most?

To solve this problem, for each pair of intersections within 10 miles of each other, calculate the increase in the flow between Juneau and Fairbanks if you add a pipe between the intersections. Each of these sub-problems is exactly maximum flow.

Back to USACO Training Gateway | Comment or Question