Introduction to Network Flow

Alex Chen

March 2, 2012

1 Introduction

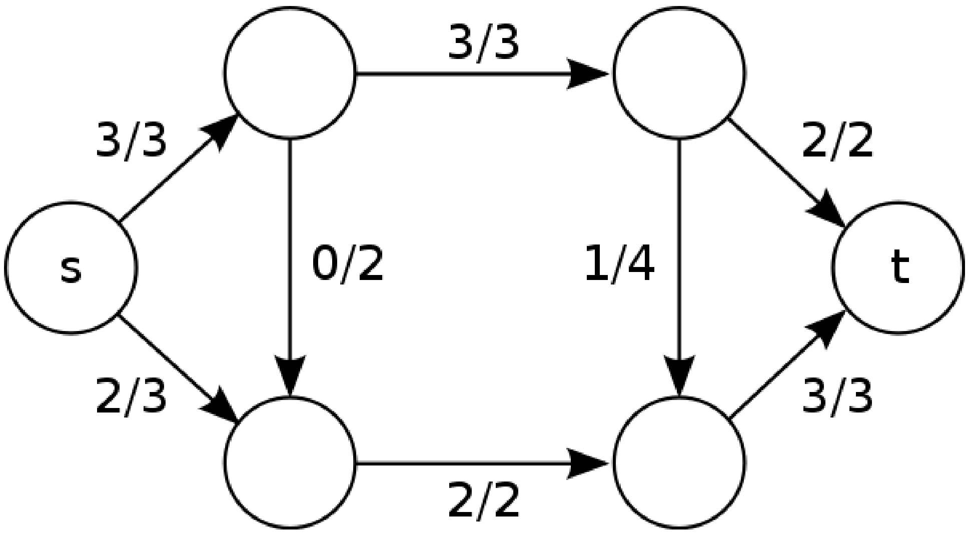
Network flow problems are never seen out of Gold, but the algorithms used to solve these problems are among the coolest ones out there and are definitely worthwhile to learn. Aside from their main application to the *maximum flow* problem, network flow is useful for a huge plethora of things, from minimum cuts to matching to others.

These problems are interesting in that typical network flow contest problems are rarely obvious. Thus, they come in many different forms, and it is a tricky task to decipher a network flow problem. Although the algorithm is one of the more complex ones, with practice you will realize that there is nothing scary in coding the Ford-Fulkerson algorithm. After all, the parts of the algorithm are all fairly simple.

2 The Maximum Flow Problem

Suppose we have a network (a graph, possibly directed) of pipes. Water can flow through the pipes. All water begins at a certain node, called the *source*, and aims to flow to another node, called the *sink*. Given the capcities of each pipe (in water per unit time), what is the maximum amount of water that can flow to the sink per unit time? The pipes are all directed (water can only flow in one direction through the pipe).

Problems like this are network flow problems. The goal is to find the *maximum flow*, the greatest possible rate at which water can reach the sink.



Note the graph shown (image from Wikipedia). The s node is the source, and the t node is the sink. The first number on each edge is the amount of water (per unit time) flowing through the edge, and the second number on each edge is its capacity. Note the way the flow is distributed and limited by the edges to lead the maximum amount of water possible to the sink.

3 Ford-Fulkerson Algorithm

One way to find the maximum flow through a graph is with the *Ford-Fulkerson algorithm.* There exist other algorithms, such as Dinic's algorithm and the push-relabel algorithm. They each have their own advantages, but Ford-Fulkerson is definitely one of the easier ones to code, and for the most part, is efficient enough to solve all problems. The pieces to Ford-Fulkerson are all simple parts: a search through a graph and some backtracking.

3.1 Augmenting Paths

An *augmenting path* is one that starts at the source, ends at the sink, and goes through edges with positive capacities. As long as we can find augmenting paths in a flow network, the current flow can be increased. The Ford-Fulkerson algorithm, like others, repeatedly finds augmenting paths and then adds them to the flow. When there are no more augmenting paths, the maximum flow has been found. However, after each augmenting path has been found, the graph must be modified (which is why the same path cannot be used multiple times).

How do we find an augmenting path? There are simple ways. One is to perform a depth-first search from the source, and another way is to perform a breadth-first search. The only requirement is that only edges with positive capacities as used. When finding this path, keep track of the edges used.

The amount of flow going through this augmenting path is equal to the smallest capacity of any edge that the path uses. This is the bottleneck edge. Let's call this flow amount F.

3.2 Backtracking

After finding an augmenting path, backtracking must occur. In order to prevent more flow from going through an augmenting path, we take the flowing amount F and push it backwards. Then, we subtract F from the capacity of each edge on the augmenting path and then for each edge on the path, we add a reverse edge with capacity F. To interpret this intuitively, we add backwards edges to allow us to reverse the direction of flow when necessary. This is in case the flow was a bad one, one that is not part of the actual solution. The backflow lets us correct for mistakes.

Note that when implementing this algorithm, it is wise to keep track of the reverse edge for each edge in the graph. That way, it is easy to access reverse edges.

By repeatedly finding augmenting paths and then backtracking, the maximum flow can be calculated.

3.3 Pseudocode

```
def find_a_path(start, end):
        // use a breadth-first search from the source to the sink
        // but only use edges with a positive capacity
        // store the edges of the path as a global variable
        return whether a path has been found

def ford_fulkerson(graph, edges, source, sink):
        max_flow = 0
        while find_a_path(source, sink):
                current_flow = minimum capacity of all edge in the path
                for each edge e in the path:
                        e.capacity -= current_flow
                        e.reverse.capacity += current_flow
                max_flow += current_flow
        return max_flow
```

3.4 Complexity

The runtime complexity is $O(Ef)$, where f is the flow. The search for the augmenting path takes $O(E)$ if implemented efficiently, and the search is run f times. There are times when the algorithm will run forever, but that never happens when the flow values are integers.

4 Max-Flow Min-Cut Theorem

This theorem states that the maximum flow of a graph is also equal to the minimum cut. The minimum cut of a graph, given a start and an end vertex, is the minimum weight of the edges that must be removed to make the end vertex unreachable from the start vertex. To compute the minimum flow using a network flow algorithm, let the start vertex be the source and the end vertex be the sink. Then, compute the maximum flow.

To find the removed edges themselves, perform a depth first search from the source using reversed versions of all the remaining edges (after running Ford-Fulkerson's).

An intuitive interpretation of the flow network graph is to see the capacity of each edge as the cost to remove it.

5 Bipartite Matching

Network flow algorithms are useful for bipartite matching. To illustrate, we can use an example. Suppose we have a list of cows and a list of food types. There is only one piece of food available for each food type. Each cow has a list of preferred food types. Bipartite matching consists of finding the maximum number of cows that can be satisfied with the limited amounts of food.

To set up the network flow graph, create a node for each cow and for each food. For each cow and food pairing, draw an edge from the cow to the food. Create a source node and connect it to every cow. Create a sink node and connect every food to it. All edges have capacity 1. The maximum flow through this graph is the *maximum matching*, the maximum pairs of cows and foods that can be matched up.

Intuitively, the edges from the cows to the foods limit each foot and cow usage in pairings to 1.

6 Problems

A lot of these problems do not look like network flow problems at first, which is what makes them tricky. See if you can figure out the graph to make for each problem. The second hardest part of a network flow problem is coding the network flow itself (so try this when you get home!).

- (Training pages) There are trucks that deliver milk between warehouses. Each truck has a cost to shut down. Find the minimum cost to prevent milk from being delivered from warehouse A to warehouse B.
- (Training pages) Given a graph, find the minimum number of *nodes* to destroy before the sink is unreachable from the source.
- There are cows and stalls, and each stall has a capacity. Each cow only likes a certain set of stalls. Find the maximum number of cows that can be satisfied without overflowing any stalls. Also, try

solving the problem for what would happen if cows can make clones of themselves (clones have the same preferences as the originals).

- (Nov11) Given a list of horizontal and vertical line segments, find the maximum set of segments such that no two intersect. No segments overlap each other.
- (Nov09) The cows are part of study groups, and each study group requires exactly one cow to bring in cookies for the group. Each cow limits the number of meetings it is willing to bring cookies for. The cows wish to divide up bringing cookies as fairly as possible. Find which cow brings cookies to which meeting.
- (Nov05) A rectangular grid has some marked cells (asteroids). Zapping a row destroys all asteroids in the row. Zapping a column destroys all the asteroids in the column. Find the minimum number of zaps to destroy the asteroids.