Network Flows

Imagine that you are a courier service, and you want to deliver some cargo from one city to another. You can deliver them using various flights from cities to cities, but each flight has a limited amount of space that you can use. An important question is, how much of our cargo can be shipped to the destination using the different flights available? To answer this question, we explore what is called a **network flow** graph, and show how we can model different problems using such a graph.

A **network flow** graph G=(V,E) is a **directed** graph with two special vertices: the source vertex s, and the sink (destination) vertex t. Each vertex represents a city where we can send or receive cargo. An edge (u,v) in the graph means that there is a flight that flies directly from u to v. Each edge has an associated **capacity**, always finite, representing the amount of space available on this flight. For simplicity, we assume there can only be one edge (u,v) for vertices u and v, but we do allow reverse edges (v,u). Firgure 1 is an example of a network flow graph modelling the problem stated above.

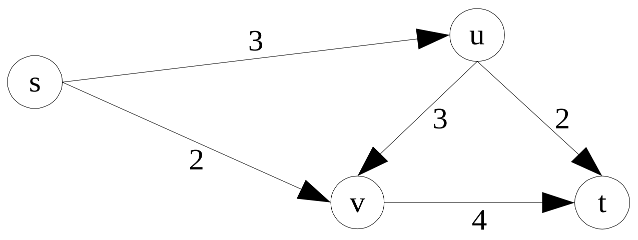

Figure 1. A simple capacitied network flow graph.

With this graph, we now want to know how much cargo we can ship from s to t. Since the cargo "flows" through the graph from s to t, we call this the **maximum flow problem.** A straightforward solution is to do the following: keep finding paths from s to t where we can send flow along, send as much flow as possible along each path, and update the flow graph afterwards to account for the used space. The following shows an arbitrary selection of a path on the above graph.

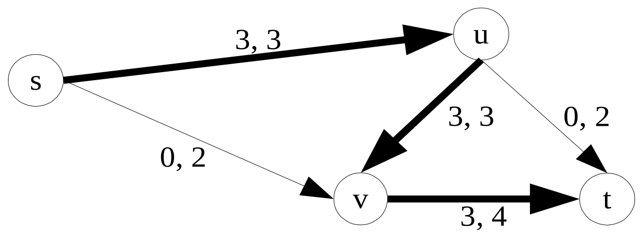

Figure 2. The first number on each edge is the flow, and the second is the capacity.

In Figure 2, we picked a path s → u → v → t. The capacities along this path are 3, 3, 4 respectively, which means we have a bottleneck capacity of 3 – we can send at most 3 units of flow along this path. Now we send 3 units of flow along this path, and try to update the graph. How should we do this? An obvious choice would be to decrease the capacity of each edge used by 3 – we have used up 3 available spaces along each edge, so the capacity on each edge must decrease by 3. Updating this way, the only other path left from s to t is s → v → t. The edge (s,v) has capacity 2, and the edge (v,t) now has capacity 1, because of a flow of 3 from the last path. Hence, with the same update procedure, we obtain Figure 3 below.

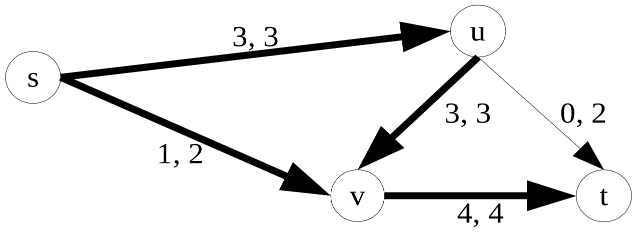

Figure 3. Using path s ➔ v ➔ t. Algorithm ends, but this is not optimal.

Our algorithm now ends, because we cannot find anymore paths from s to t (remember, an edge that has no free capacity cannot be used). However, can we do better? It turns out we can. If we only send 2 units of flow (u,v), and diverge the third unit to (u,t), then we open up a new space in both the edges (u,v) and (v,t). We can now send one more unit of flow on the path s ➔ v ➔ t, increasing our total flow to 5, which is obviously the maximum possible. The optimal solution is the following:

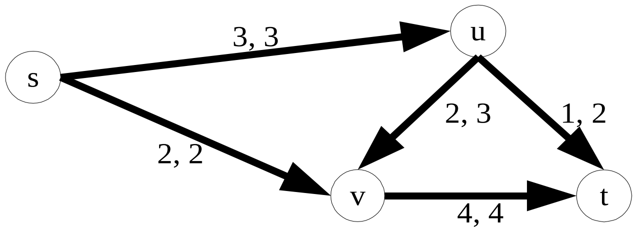

Figure 4. Optimal solution.

So, what is wrong with our algorithm? One problem was that we picked the paths in the wrong order. If we had picked the paths s ➔ u ➔ t first, then pick s ➔ u ➔ v ➔ t, then finally s ➔ v➔ t, we will end up with the optimal solution. One solution is to always pick the right ordering or paths; but this can be difficult. Can we resolve this problem without worrying about which paths we pick first and which ones we pick last?

One solution is the following. Comparing Figure 3 and Figure 4, we see that the difference between the two are in the edges (s,v), (u,v) and (u,t). In the optimal solution in Figure 4, (s,v) has one more unit of flow, (u,v) has one less unit, and (u,t) has one more unit. If we just look at these three edges and form a path s ➔ **v** ➔ **u** ➔ t, then we can interpret the path like this: we first try to send some flow along (s,v), and there are no more edges going away from v that has free capacity. Now, we can **push back** flow along (u,v), telling others that some units of flow that originally came along (u,v) can now be **taken over** by flow coming into v along (s,v). After we push flow back to u, we can look for new paths, and the only edge we can use is (u,t). The three edges have a bottle-neck capacity of 1, due to the edge (s,v), and so we push one unit along (s,v) and (u,t), but push **back** one unit on (u,v). Think of pushing flow backwards as using a backward edge that has capacity equal to the flow on that edge.

It turns out that this small fix yields a **correct solution** to the maximum flow problem. We state the algorithm formally here. We first associate a **flow** function along each edge f(u,v), that tells us how many units of flow go from u to v. Obviously, f(u,v) ≤ capacity(u,v), and we initially set f(u,v) to 0 for all edges. Now, we keep finding paths from s to t, using only two types of edges. First, we can use any edge (u,v) that has f(u,v) < capacity(u,v) [a forward edge], and second, we can use any edge (v,u) to go backward from u to v if f(v,u) > 0 [a backward edge]. If both options exist, we can pick any one of them (although picking backward edges over forward edges would avoid a lot of troubles later on). The algorithm ends when no more paths are found.

The next step is to send flow along this path. We first need to calculate the bottleneck capacity. For forward edges, this amount is just capacity(u,v)–f(u,v) – the spaces remaining. But, for backward edges, this amount is f(v,u) – the amount of flow that we can push back. Once we have the bottleneck capacity (the minimum along the path), we then send this much flow along the path, and update the graph. When updating the graph, we simply increase flow on forward edges, and decrease flow on backward edges. Here is our implementation.

Example 1. Implementation of Maximum Flow.

```
int fordFulkerson( int n, int s, int t )
{   // ASSUMES: cap[u][v] stores capacity of edge (u,v).  cap[u][v] = 0 for no edge.

    // Initialize the flow network so that fnet[u][v] = 0 for all u,v
    int flow = 0;  // no flow yet
    while( true ) {
        // Find an augmenting path, using BFS
        for( int i=0; i < n; i++ ) prev[i] = -1;
        queue< int > q;
        prev[s] = -2;
        q.push( s );
        while( !q.empty() && prev[t] == -1 ) {
            int u = q.front();
            q.pop();

            for( int v = 0; v < n; v++ ) {
                if( prev[v] == -1 ) {   // not seen yet
                    if ( fnet[v][u] || fnet[u][v] < cap[u][v] ) {
                        // either a backward edge (v,u) or a forward edge (u,v)
                        prev[v] = u;
                        q.push( v );
                    }
                }
            }
        }

        // See if we couldn't find any path to t (t has no parents)
        if( prev[t] == -1 ) break;

        // Get the bottleneck capacity
        int bot = INT_MAX;
        for( int v = t, u = prev[v]; u >= 0; v = u, u = prev[v] ) {
            if ( fnet[v][u] ) // always use backward edge over forward
                bot = min( bot, fnet[v][u] );
            else // must be a forward edge otherwise
                bot = min( bot, cap[u][v] – fnet[u][v] );
        }

        // update the flow network
        for( int v = t, u = prev[v]; u >= 0; v = u, u = prev[v] ) {
            if ( fnet[v][u] ) // backward edge -> subtract
                fnet[v][u] -= bot;
            else // forward edge -> add
                fnet[u][v] += bot;
        }

        // Sent 'bot' amount of flow from s to t, so update flow
        flow += bot;
    }

    return flow;
}
```

There are several things to note in the implementation above. We are using a BFS to find an augmenting path in each step. We set prev[s] = -2, so that it is considered seen (not seen is = -1), and its parent is not a vertex. We use this in the loops below to check for bottleneck capacity by repeatedly picking edges (u,v) in the path as long as u is a valid vertex. Also, notice that prev[] does not tell you whether a forward or backward edge is chosen. It doesn't matter, because in both cases we are going from u to v. However, when calculating the bottleneck capacity, we assume that if a backward edge exists, then we will always use it over a forward edge. This avoids the problem of flow going in both directions between vertices (why is this true?).

This sums up our algorithm. This algorithm is frequently called the Ford-Fulkerson Algorithm after its discoverers. Now, how do we know that it works? We have to prove two things: 1) that the algorithm terminates after a finite number of computations, and 2) that the algorithm outputs the maximum flow when it terminates. The proof of correctness is difficult, and is proved with the much celebrated Max-flow Min-cut Theorem that can be found in the Big White book (Cormen *et al.*, ¨Introduction to Algorithms¨) and covered in CS420, hence we will not discuss it here. However, we will prove that the algorithm always terminates.

Proof of Termination of the Ford-Fulkerson Algorithm.
When we find a path from s to t, we can only use forward edges and backward edges. Since we are finding a path, we never visit a vertex twice. Since each path goes from s to t, we will **never** find a backward edge from s, or a backward edge to t – if they exist, then we have flow coming into s or flow coming out of t, both impossible when all our paths go from s to t. Therefore, each path will **augment** the flow of the graph by at least one unit, because both the first and last edge used in the path are forward edges. Since each capacity is finite, the algorithm can only find a finite number of **augmenting paths**, and thus must end in a finite number of iterations. ■

Time complexity
The running time of the Ford-Fulkerson algorithm is O((n+m)*F), where m is the number of edges, and F is the maximum flow in the graph. This is because we can do at most F augmenting paths, each being a BFS, and so needs O(n+m) time (Our implementation of BFS above is only $O(n^2)$ because it uses an adjacency matrix). However, Edmonds and Karp proved that if we always find the **shortest** augmenting path, then we can achieve a better bound. Since BFS already finds the shortest path in an unweighted graph, the implementation above is already the Edmonds-Karp algorithm. This runs in $O(m^2 n)$ time with an adjacency list and $O(m n^3)$ with an adjacency matrix, as above, because in this case BFS takes quadratic time. There are better, more complicated algorithms that solve the maximum flow problem in $O(m n^2)$ and even $O(n^3)$ time. Descriptions of these algorithms and a proof of the $O(m^2 n)$ running time bound for the Edmonds-Karp algorithm can be found in the Big White book.

Additional Applications of Maximum Flow
Many problems in the real world or in mathematics lend themselves readily to be solved using maximum flow. "Real" networks, like the Internet or electronic circuit boards, are good examples of **flow networks** (badwidths as capacities). Many transportation problems are also maximum flow problems. In the next section we will discuss one common application of maximum flow – matching.

Bipartite matching

A *bipartite* (or *bicolourable*) graph is a graph G=(V,E) in which it is possible to split the set of vertices, V, into two non-empty sets – L and R – such that all of the edges in E have one endpoint in L and the

other one in R. A *matching* in any graph is any subset, S, of the edges chosen in such a way that no two edges in S share an endpoint. A *maximum matching* is a matching of maximum size. The problem of finding a maximum matching in a given graph is NP-hard in general, but if the graph is bipartite, there is a neat polynomial time solution.
First of all, why do we care about bipartite matching? Here is a classic problem. There are m job applicants and n job openings. Each applicant has a subset of the job openings she is interested in. Conversely, each job opening can only accept one applicant out of some subset of the applicants. In other words, there are certain allowed or "compatible" pairings of applicants to jobs. Find an assignment of jobs to applicants in such a way all the pairings are allowed and as many applicants as possible get jobs.

We can model this with a bipartite graph – on the left side we have the applicants and on the right side we have the jobs. Draw an edge between applicant u and job v if they are compatible. The answer that we are looking for is the maximum matching.

One way to solve the problem is to reduce it to an instance of Maximum Flow. Draw the bipartite graph with applicants on the left and jobs on the right. Give each edge a capacity of 1 – having a flow of 1 between applicant u and job v will correspond to matching applicant u to job v. Now we need to ensure that no applicant gets matched to more than 1 job. For that, add a vertex, s, on the far left and connect it each applicant by an edge of capacity 1. We also want each job to only be filled by at most one applicant. To enforce that, add a vertex, t, on the far right and connect each job opening to t by an edge of capacity 1. Finally, find the maximum flow from s to t. The amount of flow we can push is exactly the number of original edges that will be used to connect an applicant to a job. Furthermore, no applicant or job vertex will be used more than once. To get the optimal matching, read it off from the resulting flow network.

The implementation of this algorithm is straightforward once we have the code for MaxFlow. However, since each edge has capacity 1, we can simplify the code a lot. Here is an implementation that uses DFS to find augmenting paths in the Ford-Fulkerson algorithm and represents the graph and the flow network implicitly.

Example 1: Maximum Bipartite Matching

```
// define M and N to be the maximum sizes of the left and right set respectively
bool graph[M][N];
bool seen[N];
int matchL[M], matchR[N];
int m, n;

bool bpm( int u ) {
    for( int v = 0; v < n; v++ ) if( graph[u][v] ) {
        if( seen[v] ) continue;
        seen[v] = true;

        if( matchR[v] < 0 || bpm( matchR[v] ) ) {
            matchL[u] = v;
            matchR[v] = u;
            return true;
        }
    }
    return false;
}

int main() {
    // Read input and populate graph[][]
    // Set m to be the size of L, n to be the size of R
    memset( matchL, -1, sizeof( matchL ) );
    memset( matchR, -1, sizeof( matchR ) );
    int cnt = 0;
    for( int i = 0; i < m; i++ )
    {
        memset( seen, 0, sizeof( seen ) );
        if( bpm( i ) ) cnt++;
    }

    // cnt contains the size of the matching
    // matchL[i] is what left vertex i is matched to (or -1 if unmatched)
    // matchR[j] is what right vertex j is matched to (or -1 if unmatched)
    return 0;
}
```

The code is surprisingly short, compared to the maximum flow code, and most of the work is done by the bpm(u) function that tries to match the left vertex u to something on the right side. It does that by trying all right vertices v and assigning u to v if either v is unassigned, or if v's match on the left side can be reassigned to some other vertex on the right that is larger than v. If you try running this algorithm on an example, you will see that this is simply another way of implementing DFS. Each call to bpm(u) finds an augmenting path starting at u. The path is allowed to use left-to-right edges that are still unused, as well as right-to-left edges that are used (by undoing a unit of flow). bpm(u) returns true if a path was found and false otherwise. Each augmenting path adds one more edge to the matching, and these are counted by the 'cnt' variable in main(). The seen[] array is the usual DFS seen [] array. matchL[] and matchR[] store the matching; a -1 means "vertex is not matched (yet)".

Can you determine the worst-case running time of this algorithm? (Typing the link below in the browser and reading it there would be cheating.) Note that the bipartite matching problem is symmetric. We can flip the left and right sides of the graph and assign jobs to applicants instead of applicants to jobs – it's the same thing. Using this fact, can you improve the worst-case running time? How could you do that with the fewest changes to the code?

This code, with a lot more comments can be found here:
http://www.lexansoft.com:8081/tools/bpm.cpp

And here is an implementation of maximum flow:
http://www.lexansoft.com:8081/tools/flow.cpp

References

- Cormen, Thomas H., et al. Introduction to Algorithms. , 2nd ed. Cambridge: The MIT press, 2002.