Maximum Flow

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).

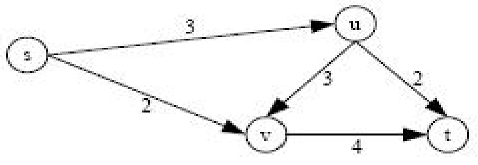

Figure 1. A simple capacitied network flow graph.

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.

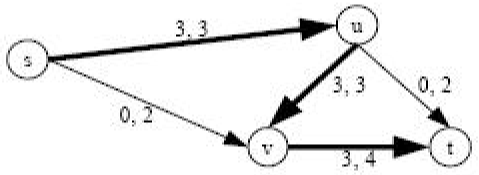

Figure 2. The first number on each edge is the flow, and the second is the capacity.

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.

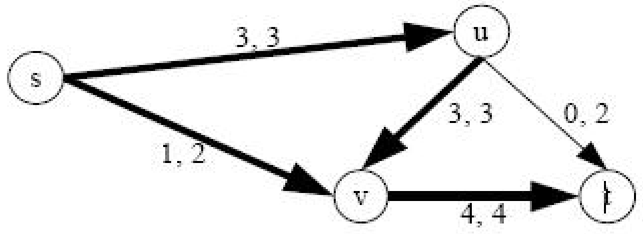

Figure 3. Using path s → v → t. Algorithm ends, but this is not optimal.

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:

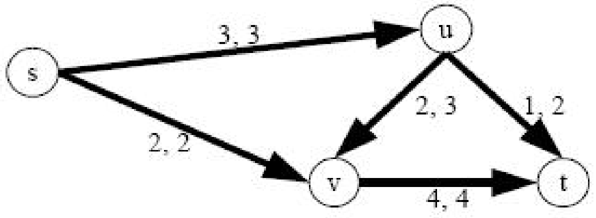

Figure 4. Optimal solution.

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), (v,u) 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 bottleneck capacity of 1, due to the edge (s,v), and so we push one unit along (s,v) and (u,t), but push **bac**k 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 **capacity** function along each edge c(u,v), that tells us how many units of flow can go from u to v. Initially, c(u, v) is set to the maximum capacity for each edge (u, v). Now, we keep finding paths from s to t (we refer to these paths as **augmenting paths**). When an augmenting path is found, we adjust the capacity for each edge in the path. The algorithm ends when no more paths are found.

More specifically, each augmenting path we find will have an associated 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

```
#include <iostream>
#include <algorithm>
using namespace std;

#define NV 4                      // the number of vertices in our graph

int capacity[NV][NV];             // our flow graph
bool seen[NV];                    // seen array for DFS

int findAugPath(int currentNode, int sinkNode, int flowAmt) {
      // if we've reached the sinkNode, then we have found an augmenting path
      if (currentNode == sinkNode) return flowAmt;

      // mark current node as seen
      seen[currentNode] = true;

      for (int nextNode = 0; nextNode < NV; nextNode++)
            if (!seen[nextNode] && capacity[currentNode][nextNode]) {

                  // calculate how much flow we can use if we take this edge
                  int amt = min(flowAmt, capacity[currentNode][nextNode]);

                  // see if we can find an augmenting path that uses nextNode
                  // f will contain the amount of flow that this path can handle
                  // (f == 0 if no path exists)
                  int f = findAugPath(nextNode, sinkNode, amt);

                  // if we found a path, update the capacity network and return
                  // the amount of flow in the augmenting path
                  if (f) {
                        capacity[currentNode][nextNode] -= f;
                        capacity[nextNode][currentNode] += f;
                        return f;
                  }
            }

      // no augmenting path was found, return 0
      return 0;

}

int main() {

      // initialize capacity[][] with  appropriate values
      memset(capacity, 0, sizeof(capacity));
      capacity[0][1] = 3;
      capacity[0][2] = 2;
      capacity[1][2] = 3;
      capacity[1][3] = 2;
      capacity[2][3] = 4;

      // search for augmenting paths
      int n = 0, flow = 0;
      do {
            flow += n;
            memset(seen, false, sizeof(seen));
            n = findAugPath(0, 3, INT_MAX);
      } while (n);

      cout << "Maximum flow: " << flow << endl;

      return 0;
}
```

There are several things to note in the implementation above. We are using DFS to find augmenting paths in each step. The beauty of using DFS is that it makes it very easy to keep track of the bottleneck capacity – each time a new edge (u, v) is added to the path, the bottleneck capacity becomes the minimum of the current bottleneck capacity and the capacity of the edge (u, v). As soon as an augmenting path with capacity **C** > 0 is found, we adjust our capacity matrix and return **C**.

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 algorithm given above is $O(n^2 F)$, where F is the maximum flow in the graph. This is because there will be at most F augmenting paths, and finding an augmenting path via DFS takes $O(n^2)$ time (re-writing our DFS to use an adjacency list would change the time complexity to $O(mn*F)$, where m is the number of edges in the graph; this would be an advantage if the capacity graph were sparse).

Edmonds and Karp proved that if we always find the **shortes**t augmenting path, then we can achieve a better bound; specifically, if we change our DFS to BFS then we will achieve a running time of $O(m^2 n)$. Furthermore there are better, more complicated algorithms that solve the maximum flow problem more efficiently.

Additional Applications of Maximum Flow

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

Additional sample code can be found here:
http://www.shygypsy.com/tools/

References

Cormen, Thomas H., et al. Introduction to Algorithms. , 2nd ed. Cambridge: The MIT press, 2002.
Igor & Frank's notes (thanks guys!)