The Ford Fulkerson Algorithm

Victor Wu and Matt Price

State University of New York, Binghamton University

CS375, *Design and Analysis of Algorithms*

Dr. Lijun Yin

December 22, 2015- *Fall 2015*

**Objective**

For this project, Victor and I decided to show and implement the Ford Fulkerson algorithm for calculating maximum flow in a flow network. We were interested in this algorithm for a variety of reasons. Firstly, it is quite applicable in the real world. A flow network is essentially a network of nodes, or a graph, that has a starting point and a finishing point. The starting pointing is termed the start node, and the finishing point the sink node. The objective is to find the maximum amount of things that can be transferred along paths from source to sink. There may be multiple paths, called augmenting paths, and each edge along the path carries a flow and a capacity value. It will be easier to understand flow by defining capacity first. Capacity is the limit of things you can pass through each edge. Setting up pipes for a certain infrastructure like a home is a solid example of a flow network. We have a main reserve where the water comes from, whether it be underground or through a piping mechanism linked to a sanitation system. We have to send a certain amount of fluids to the faucet in the sink or the tub, without hitting full capacity. There may be many pipes in this structure, and many paths from the source to the sink. Each of these pipes has a flow which cannot exceed the capacity. We start by picking a path that will get us to our destination. Each pipe will have a certain capacity because it is different in size. Let's take an example of a path with values and see how this works.

We have 4 pipes: source to A, A to B, B to C and C to sink. Let's call them pipe1, pipe2, pipe3, and pipe4, respectively. Pipe1 has a capacity of 10, pipe2 a capacity of 8, pipe3 a capacity of 9 and pipe 4 a capacity of 10. Before we start sending water down pipe1, we need to figure out how much water to send. This is where residual capacity, or bottleneck capacity, comes in. We pick the pipe with the lowest capacity, and send that much water down pipe1. So, in this case, out bottleneck capacity is 8. We send 8 units of water down pipe1, pipe2, pipe3, and pipe4. Keep in mind that each pipe has 2 values, the flow and the capacity. The flow now becomes 8/10, 8/8, 8/9 and 8/10. The sink node receives 8 units of water, and we update our max flow to 8. Depending on how many paths exist from source to sink, we repeat this process for each path until we no longer have augmenting paths. We have no more augmenting paths when the capacity is filled up somewhere along the path, or we've reached a block. We update the max flow on each path iteration. The max flow should add up to the 2 edges coming out of the source or the 2 edges coming into the sink. Other real world applications include modeling traffic on the roads, or moving currents in a circuit.

***Pseudo Code***

The pseudocode for this algorithm will better explain its implementation. Let us examine the following algorithm (Wikipedia):

Input

Given Network G =s (V, E)

Flow capacity c, source node s, sink node t, path p

Output

# ƒ(u, v) -> 0 ∀ (u, v)

# ∃p from s to t in Gf such that cf(u,v) > 0 ∀ (u, v) ∈ p

# cf(p) = min { cf(u, v) : (u, v) ∈ p }

# ∀ (u, v) ∈ p:

## ƒ(u, v) -> ƒ(u, v) + cf(p)

## ƒ(v, u) -> ƒ(v, u) - cf(p)

In other words, we have a directed network, or directed linear graph, G consisting of a collection V of elements, x, y, etc. with a subset E of the ordered pairs (x, y) of elements taken from V. V could analogically thought of as nodes, vertices, junction points, or points; while members of E are arcs, links, branches, or edges. So we are given a graph G, a flow capacity, one source node, one sink node, and a path. We start off by initializing the flow to be 0 for all edges. And while there is a path in Graph G where the capacity is greater than 0 for each edge, we calculate the very minimum capacity by running a Breadth-First Search (BFS) or Depth First Search (DFS). The variation of using the latter, DFS, is essentially the core of a spin-off of this algorithm, the Edmonds-Karp algorithm, which has a (E^2 \* V) running time. After placing them on a stack, we check each of the edges for the capacity, which we can store in an adjacency matrix or a linked list. We let f be the minimum capacity value along p, and we add f to the max flow. For each edge(u,v) on p, we add f to the flow. Note that BFS can also be used to pick paths as well.

Now we'll delve into the interesting part: the code behind our implementation. We did this in c+++ to avoid reconstructing queues with their individual queue operations. We start off by defining the number of nodes we will use in the graph. We have 2 main functions, one for breadth first search and the other for Ford Fulkerson itself. BFS takes in some important parameters: an adjacency matrix for the actual residual graph, the source node, the sink node, and an array for storing the nodes in the path. As per usual BFS implementation, we created a visited array that marks the nodes as visited or not, and initialized the vertices to 0. We then declared a queue, pushed the source node into the queue, and marked the source node as visited. We set the parent of s to -1 because it is the first node. We run through a typical BFS loop where we iterate through the vertices and push new nodes into the queue, and store their parent nodes, or the old nodes, in the parent array. We mark each node traversed as visited. If the BFS leads us to the sink, then we return true, and false otherwise. So this function just sets up a way to find each path from source to sink.

The actual meat of the problem is finding the maximum flow from source to sink. This is accomplished in our Ford Fulkerson algorithm. It takes similar parameters, but instead of taking the residual graph, we take in the regular graph that we define in an adjacency matrix in main(). We also take in the source and sink nodes. We declare the residualGraph[numVertices][numVertices], which basically just says if there is a bottleneck capacity(residual capacity) between the two nodes. Like a regular adjacency matrix, if residualGraph[u][v] is 0 for example, then there is no bottleneck capacity. If 1, then there is. We again declare the parent array with the number of vertices inside to store the path. We also initialize max flow to 0.

Having already created a BFS algorithm for finding paths, we can now use it to augment the actual paths. While BFS is active, we find the minimal bottleneck capacity of the edges along the path. So, we made a path flow variable and set it to the max flow that we can possibly get. Here's where it gets tricky. After we traverse through a possible path with BFS, we need to backtrack through the path to find the minimum bottleneck capacity to use. So we iterate from sink back to the source, going through each value in the parent array, and setting path flow to the minimum value of each edge in the residual graph. Remember that our residual graph just stores the edges, so we are just comparing each edge to the parent edge before it in the path. Once the bottleneck capacity is found, we simply add it to each current edge in the path from sink to source. Once each edge has been updated, we add the path flow to our max flow variable with was originally set to 0. Now it will show the maximum flow along the path. We exit out of the while BFS loop and return the max flow.

In our main function we have a simple adjacency matrix called graph[numvertices][numVertices] that takes in the capacity of each edge between the two nodes. It's 0 if there's no connection, and an actual value otherwise.

We originally had used vectors instead to store the nodes in the path, but the implementation was a bit trickier and we were encountering constant errors. We also went between taking input from a text file in the format of *node u, node v*, and taking user defined input as to how many nodes he/she wanted to use. The second option seemed like the most viable and efficient. We would ask the user how many nodes to enter, and from that input, make an automatically generated adjacency matrix. The thing is, that required more hardcoding into the main function. We also were thinking about generating the actual visual representation of the graph in the output, without the flow and capacity of course. So accompanied with the maximum flow, we'd see how many nodes and edges the graph had. That would have been pretty sleek, but we went against it due to time constraints.

Our algorithm performed in O(max flow \* E) time, because we iterate through each of the edges in the path and add flow in every iteration. Note that there are other ways to traverse through the paths, namely depth first search. This would generate the same time complexity, but instead of finding different paths, it would just go all the way down one. So, we thought BFS was better in this case.

We noticed that there are definitely ways to improve the algorithm. There are variants of Ford Fulkerson that are polynomially solvable, and where the augmenting path can be chosen cleverly. Firstly, augmenting along paths with maximum capacity can be more efficient. This can be implemented in the following way. Assuming the bottleneck capacity is known, we start by removing all the edges with residual capacity less or equal to it. We then check if there is a path from source to sink, and we binary search the largest bottleneck capacity. This has a performance of O(max flow log Edge).

Secondly, augmenting along the shortest path(the shortest number of edges) would also lead to faster running times. To find these paths, we'd still have to use BFS to compare which paths have less edges. This would give us a performance of O(m\*n), with m being the time to find the shortest path via BFS and n the amount of augmentations.

Naturally, finding maximum flow requires a greedy approach. We are finding a path where the flow is less than the capacity for each edge along the path. The problem with the greedy approach though is that we can get stuck along the same path, because it always looks for the flow less than capacity.

***Actual Implementation***

After careful analysis of the algorithm, and a better understanding, we implemented this algorithm in C++. As mentioned in the README file located in the project solution, there was a minor segmentation fault error. Debugging in Valgrind, with suppressed 3 errors, leads to a successful execution of the program, and verifying our logic is correct for the most part – in addition to the black-box testing we have done. We would have loved to do more white-box testing through GDB, but found that convenient break points didn’t help us much with the information we needed to resolve the segmentation error.

We represented an abstract tree using vector data structures, through a vector of edges and nodes. Doing our BFS search, we have also used a queue data structure as per the standard implementation. The source code has been properly and extensively, by de facto standards, documented so the actual implementation could be reviewed for deeper understanding.

***Results***

The results are self-evident in the sense that there is a segmentation fault, and while we believe it is a simple error that we overlooked, it has proven to be far more difficult to debug than we expected. It made the time-constraint appropriate for a problem like this more demanding, and we decided to use Valgrind, with suppressed errors, to successfully retrieve the output of this program. We believe that our implementation of our program was rewarding, and the logic behind the algorithm has been successfully implemented.

Since we are unable to offer exact testing measurements since Valgrind is limited, we would like to talk more about the practical usage of this algorithm. Since the run-time is bounded by O(Eƒ), it is efficient in most network flow problems. Consider, in optimization theory, the max-flow min-cut theorem, in which a flow network having a maximum amount of flow passing from source to sink is equal to the minimum capacity in which, when removed from the network in a particular way, causes no flow to be passible from source to sink. Using the Ford-Fulkerson algorithm, we can easily compute the flow in this otherwise difficult proof.

The results of our research and implementation of the Ford-Fulkerson algorithm lead us to have a deeper understanding of network flows, and we believe our results demonstrates how practical network flows are in our lives, and the importance for solving for these flows.

**References**

Ford, L., & Fulkerson, D. (2010). Flows in networks. Princeton, N.J.: Princeton University

Press.

Ford–Fulkerson algorithm. (2015, November 11). In Wikipedia. Retrieved from

https://en.wikipedia.org/wiki/Ford–Fulkerson\_algorithm