Maximum Capacity Flow Augmentation

Amos Fiat[*] *Shai Vardi*[*]

January 11, 2012

1 Variants of the Ford Fulkerson network flow algorithm

Let $G = (V, E), s, t$, and $c : V \times V \mapsto \Re^+$ be a flow network and let f^* be some maximal flow in this network. The residual graph relative to some flow f is labeled G_f. In this section we give a general overview of what we covered in max flow algorithms and mention the main ideas. In the next section we will give a detailed proof for the *max capacity augmenting path* algorithm, for which we could not find a good reference.

We considered the following variants of the Ford Fulkerson algorithm in class:

1. If the capacities are integers and one chooses an arbitrary augmenting path in the residual network, the number of path augmentations during the execution of the Ford Fulkerson algorithm is $O(|E||f^*|)$ and there are cases where this is tight. For example, see Figure 1.

2. If the capacities are integers and one chooses an augmenting path of max capacity in the residual graph, *i.e.*, an $s - t$ path whose capacity is maximal out of all the $s - t$ paths, (the path capacity is the minimal capacity of an edge along the path), then the number of path augmentations is bounded by $O(|E| \log(|f^*|))$. Also, we can find such a path of maximal capacity in time $O(|V| \log |V| + |E|)$, by a slight modification of the Dijkstra single source shortest path algorithm. The proofs are given below and are similar to proofs for "scaling max flow algorithms" [3]. This max capacity augmentation algorithm is weakly polynomial time[1], as the running time depends on the size of the integers ($\log |f^*|$ is bounded by $\log \sum_{u,v \in V} c(u, v)$).

3. The Edmonds-Karp variant of Ford-Fulkerson: choose an augmenting path in G_f to be an $s - t$ path with a minimal number of edges. The main argument about Edmonds Karp is that every time a specific edge, (u, v), is critical in G_f, it cannot be critical again until the shortest path in the residual graph increases. So if (u, v) is also critical in G'_f some time later, then the length of the shortest path from s to u in the residual graph G_f, $\delta_f(s, u)$,

[*]School of Computer Science, Tel Aviv University

[1]*Strongly polynomial time* algorithms run in time that is polynomial in the *number* of integers in the input, not in their size. An algorithm that runs in polynomial time but is not strongly polynomial time is said to be *weakly polynomial time*.

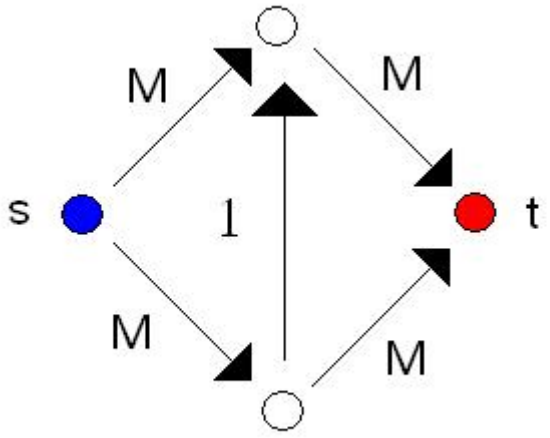

Figure 1: Diamond Network

is strictly less than the length of the shortest path from s to u in the residual graph $G_{f'}$, $\delta_{f'}(s,u)$. Because the shortest path in the graph is monotonically increasing, it follows that (u,v) can be critical no more than $O(|V|)$ times, and every augmenting path has at least one critical edge, from which we derive that the total number of augmenting paths is no more than $O(|E||V|)$. Finding a such an $s-t$ path via BFS requires time $O(|E|)$. The Edmonds Karp algorithm is strongly polynomial - it does not depend on the lengths of the integers in the input.

4. The Dinic (or Dinitz) variant of Ford-Fulkerson: Find a blocking flow in the layered graph L_f derived from the residual graph G_f. The layered graph L_f contains the edges of G_f, $(u,v) \in E_f$, for which $\delta_f(s,u)+1 = \delta_f(s,v)$. Finding a blocking flow in such a layered graph can be done in time $O(|V||E|)$. The number of augmenting paths required for a blocking flow in L_f is no more than $|E|$, and the length of such a path is no more than $|V|$. Thus, a DFS search from s will find t in time $O(|V|)$, or else it will start following edges that cannot lead to t, “wasted edges".

   However, the total time spent on such “wasted edges" is only $O(|E|)$, so long as $\delta_f(s,t)$ remains unchanged. When doing a DFS from s and searching for t, if we ever pop a vertex v from the DFS stack, and u is now at the top of the stack, then this means that that the edge $(u,v) \in L_f$ can no longer be extended to a path that leads to t (or if it did not have such a continuation to begin with), then when v is popped from the DFS stack one can remove (u,v) from L_f, this edge will not need to be traversed again when performing a new DFS from s.

   The Dinic algorithm is covered in Tarjan’s book [4] and in the original Even-Tarjan paper [2]. These sources also cover the various $0/1$ network results. Also, there’s a recent writeup by Dinitz himself giving the history of the development of the algorithm [1].

5. We also considered $0/1$ flow networks in the context of the Dinic algorithm. One difference between $0/1$ networks and other networks is that when we augment flow along an

augmenting path, all the edges of the path disappear from L_f. (The edges pointing back are added to G_f, but not the L_f). Ergo, it follows that the total time spent on augmenting flows by choosing augmenting paths in the layered graph L_f is only $O(|E|)$, for each value of $\delta_f(s,t)$, and there are only $|V|-1$ legal values for $\delta_f(s,t)$.

6. A more refined analysis shows that the time required is only $O(|E|^{3/2})$, which is better for sparse graphs. This follows by showing that the length of the path from s to t, measured by the number of edges, $\delta_f(s,t)$, depends on the max flow in G_f: if the value of the max flow in G_f is M then $\delta_f(s,t) \leq |E|/M$. This follows by considering cuts determined by taking s and successive layers in the layered graph, vs. all the rest. The max flow min cut theorem tells us that the capacity of the cut (the number of unit capacity edges) must exceed the max flow.

7. We also gave an upper bound on $\delta_f(s,t)$ as a function of the number of vertices, $|V|$, we get that $\delta_f(s,t) \leq 2|V|/M$, from this we can obtain that the time required by Dinic's algorithm with unit capacity edges is $O(|V|^{2/3}|E|)$.

2 Proofs related to the max capacity augmentation variant

To prove the max capacity augmentation variant of Ford Fulkerson, we need several lemmas. Before we prove Lemma 2.2, we will prove a simpler lemma as a warm up:

Lemma 2.1. *Let $G = (V,E), c : V \times V \mapsto \Re^+, s, t$, be a flow network. Let f_{max} be some maximum flow in G. f_{max} can be represented as the sum of $|E|$ flow paths in G.*

Proof. Let H be a group of flow paths in G, and initialize $H = \emptyset$. Choose some $s-t$ path in G which has a positive flow on it. Add this path with flow $f(e)$, to H , where e is the edge through which the flow is minimal from all the edges in the path. As e is saturated, it can no longer appear on any other path. In each iteration we saturate at least one edge, therefore $|H| \leq |E|$. □

Lemma 2.2. *Let $G = (V,E), c : V \times V \mapsto \Re^+, s, t$, be a flow network. Let f be some flow in G. Let G_f be the residual network. The max flow in the residual network G_f can be represented as the sum of $|E|$ augmenting flow paths in G_f.*

Proof. Let $f_{\max G_f}$ denote a max flow for G_f. Consider the graph, $G_{\max(f)}$ consisting only of edges of G_f used in $f_{\max G_f}$. We can assume that if (u,v) appears in this graph, then (v,u) does not. It follows that the number of edges in $G_{\max(f)} \leq |E|$ (although G_f may contain up to $2|E|$ edges). Choose some $s-t$ path in F and let the flow along this path be the minimal flow in $f_{\max G_f}$ along the path. At least one edge in $G_{\max(f)}$ will be saturated (flow equal to the residual capacity). Every such $s-t$ path will remove one edge, so $|E|$ such paths will cover the entire flow $f_{\max G_f}$. □

We now look at maximum capacity augmenting paths:

Let $G = (V,E), c : V \times V \mapsto \Re^+, s, t$, be a flow network, and let f^* be the value of the max flow in G. Let f_0 be some flow in G. Label by $c(P)$ the capacity of a path, i.e. the minimum capacity of all edges on the path. Choose a max capacity augmenting path in G_{f_0},

P_1. Augment the flow by the capacity of P_1, to get flow f_1. $|f_1| = |f_0| + c(P_1)$. Now, choose a max capacity augmenting path in G_{f_1}, P_2. Augment the flow by the capacity of P_2, and so on. Repeat $2|E|$ times, getting flows $f_0, f_1, f_2, \ldots, f_{2|E|}$, derived from max capacity augmenting paths $P_1, P_2, \ldots, P_{2|E|}$ in the residual flow networks $G_{f_0}, G_{f_1}, \ldots, G_{f_{2|E|-1}}$, respectively.

Lemma 2.3. *The maximal flow in the residual graph $G_{f_{2|E|-1}}$ is no more than $1/2$ the maximal flow in the residual graph G_{f_0}.*

Proof. The maximal flow in the residual graph G_{f_0} is equal to $f^* - |f_0|$. Therefore, by an averaging argument, it must be that for some $j = 1, \ldots, 2|E|$, the capacity of the path P_j in the residual network $G_{f_{j-1}}$ is no more than $\frac{f^* - |f_0|}{2|E|}$. But, P_j is a max capacity path, so from Lemma 2.2 the maximal flow in the residual network $G_{f_{j-1}}$ is no more than $|E|$ times the capacity of P_j, i.e., no more than $\frac{f^* - |f_0|}{2}$. □

Lemma 2.4. *The total number of augmenting paths used when choosing the max capacity augmenting path is no more than $2|E| \log(f^*)$ when starting in a flow network whose maximum flow is some integer value f^*.*

Proof. After finding $2|E|$ maximum capacity paths, the flow in the residual network decreases to at most half. Therefore we need at most $\log(f^*)$ such iterations to find the maximal flow. □

Lemma 2.5. *We can find the max capacity path in a flow network $G = (V, E)$ in time $O(|V| \log |V| + |E|)$*

Proof. This is a variant of the Dijkstra single source shortest path algorithm. There is a heap whose entries are vertices $v \in V$ and whose key, $d[v]$ is the value of the largest capacity path from s to v, known so far. Each time we do a *delete min* from the heap, removing vertex w from the top of the heap, we have already determined the final max capacity path from s to w. Initially, the $d[s] = \infty$ and for all other vertices $v \in V - \{s\}$, $d[v] = 0$. When we do a delete min, and w was at the top of the heap, we update the path capacities for all neighbors of w that are still in the heap. For every edge (w, u), u in the heap, we set

$$\text{if } d[u] < \min(d[w], c(w, u))$$
$$d[u] = \min(d[w], c(w, u))$$

(Akin to *relax* in the original Dijkstra algorithm.) Note that this is an increase key operation and can be done in amortized $O(1)$ time using Fibonacci Heaps. (Although we are doing increase key rather than decrease key, you can multiply everything by -1 and get decrease key). □

References

[1] Yefim Dinitz. Dinitz' algorithm: The original version and even's version. In *Essays in Memory of Shimon Even*, pages 218–240, 2006.

[2] Shimon Even and Robert Endre Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4(4):507–518, 1975.

[3] Jon M. Kleinberg and Éva Tardos. *Algorithm design*. Addison-Wesley, 2006.

[4] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.