Dinic's Algorithm

We follow Tarjan's presentation [1]. In the Edmonds–Karp algorithm, we continue to augment by path flows along paths in the level graph L_G until every path from s to t in L_G contains at least one saturated edge. The flow at that point is called a *blocking flow*. The following modification, which improves the running time to $O(mn^2)$, was given by Dinic [2]. Rather than constructing a blocking flow path by path, the algorithm constructs a blocking flow all at once by finding a maximal set of minimum-length augmenting paths. Each such construction is called a *phase*. As in Edmonds–Karp, there are at most n phases, because with each phase the minimum distance from s to t in the residual graph increases by at least one.

The following algorithm describes one phase. We traverse the level graph from source to sink in a depth-first fashion, advancing whenever possible and keeping track of the path from s to the current vertex. If we get all the way to t, we have found an augmenting path, and we augment the flow by that path. If we get to a vertex with no outgoing edges, we delete that vertex (there is no path to t through it) and retreat.

In the following, u denotes the vertex currently being visited and p is a path from s to u.

Initialize. Construct a new level graph L_G. Set $u := s$ and $p := [s]$. Go to **Advance**.

Advance. If there is no edge out of u, go to **Retreat**. Otherwise, let (u, v) be such an edge. Set $p := p \cdot [v]$ and $u := v$. If $v \neq t$ then go to **Advance**. If $v = t$ then go to **Augment**.

Retreat. If $u = s$ then halt. Otherwise, delete u and all adjacent edges from L_G and remove u from the end of p. Set $u :=$ the last vertex on p. Go to **Advance**.

Augment. Let Δ be the bottleneck capacity along p. Augment by the path flow along p of value Δ, adjusting residual capacities along p. Delete newly saturated edges. Set $u :=$ the last vertex on the path p reachable from s along unsaturated edges of p; that is, the start vertex of the first newly saturated edge on p. Set $p :=$ the portion of p up to and including u. Go to **Advance**.

We now discuss the complexity of these operations.

Initialize. This is executed only once per phase and takes $O(m)$ time using BFS.

Advance. There are at most $2mn$ advances in each phase, because there can be at most n advances before an augment or retreat, and there are at most m augments and m retreats. Each advance takes constant time, so the total time for all advances is $O(mn)$.

Retreat. There are at most n retreats in each phase, because at least one vertex is deleted in each retreat. Each retreat takes $O(1)$ time plus the time to delete edges, which in all is $O(m)$; thus the time taken by all retreats in a phase is $O(m + n)$.

Augment. There are at most m augments in each phase, because at least one edge is deleted each time. Each augment takes $O(n)$ time, or $O(mn)$ time in all.

Each phase then requires $O(mn)$ time. Because there are at most n phases, the total running time is $O(mn^2)$.

The MPM Algorithm

The following algorithm of Malhotra, Pramodh-Kumar, and Maheshwari [3] produces a max flow in time $O(n^3 \log n)$ using heap-based priority queues or $O(n^3)$ using Fibonacci heaps (an implementation of priority

queues that you will see in CS 6820). The overall structure of the algorithm is similar to Edmonds–Karp or Dinic. Blocking flows are found for level graphs of increasing depth. The better time bound is due to an $O(n^2 \log n)$ (or $O(n^2)$ with Fibonacci heaps) method for producing a blocking flow.

For this algorithm, we need to consider the capacity of a vertex as opposed to the capacity of an edge. Define the *capacity* $c(v)$ of a vertex v to be the minimum of the total capacity of its incoming edges and the total capacity of its outgoing edges:

$$c(v) \quad = \quad \min \sum_{u \in V} c(u, v), \ \sum_{w \in V} c(v, w).$$

Intuitively, this is the maximum amount of commodity that can be pushed through that vertex. This definition applies as well to residual capacities obtained by subtracting a nonzero flow.

The MPM algorithm proceeds in phases. In each phase, the residual graph is computed for the current flow, and the level graph L_G is computed. If t does not appear in L_G, we are done. Otherwise, all vertices not on a path from s to t in the level graph are deleted.

Now we repeat the following steps until a blocking flow is achieved:

1. Find a vertex v of minimum capacity d. If $d = 0$, do step 2. If $d \neq 0$, do step 3.
2. Delete v and all incident edges and update the capacities of the neighboring vertices. Go to 1.
3. Push d units of flow from v to the sink and pull d units of flow from the source to v to increase the flow through v by d. This is done as follows:

   Push to sink. The outgoing edges of v are saturated in order, leaving at most one partially saturated edge. All edges that become saturated during this process are deleted. This process is then repeated on each vertex that received flow during the saturation of the edges out of v, and so on all the way to t. It is always possible to push all d units of flow all the way to t, since every vertex has capacity at least d.

   Pull from source. The incoming edges of v are saturated in order, leaving at most one partially saturated edge. All edges that become saturated by this process are deleted. This process is then repeated on each vertex from which flow was taken during the saturation of the edges into v, and so on all the way back to s. It is always possible to pull all d units of flow all the way from s, since every vertex has capacity at least d.

   Either all incoming edges of v or all outgoing edges of v are saturated and hence deleted, so v and all its remaining incident edges can be deleted from the level graph, and the capacities of the neighbors updated. Go to 1.

It takes $O(m)$ time to compute the residual graph for the current flow and level graph using BFS. Using a heap-based priority queue or Fibonacci heap, it takes $O(n \log n)$ time over all iterations of the loop to find and delete a vertex of minimum capacity. It takes $O(m)$ time over all iterations of the loop to delete all the fully saturated edges, since we spend $O(1)$ time for each such edge. It takes $O(n^2)$ time over all iterations of the loop to do the partial saturations, because it is done at most once in step 3 at each vertex for each choice of v in step 1.

The only thing that we have not accounted for is that when we push flow through a vertex, we must decrement its capacity, which may change its priority in the priority queue. To update the priority of the vertex takes time $O(\log n)$ with a heap-based priority queue or $O(1)$ with a Fibonacci heap. We may need to do this once for each completely saturated edge and once for each partial saturation. In the entire phase, there are at most m complete saturations, since the saturated edge is deleted, and at most n^2 partial saturations, as argued above. Thus over the entire phase, the amount of time needed to decrement the priorities in the queue is $O(n^2 \log n)$ for a heap-based priority queue or $O(n^2)$ for a Fibonacci heap.

The loop thus achieves a blocking flow in $O(n^2 \log n)$ time with a heap-based priority queue or $O(n^2)$ for a Fibonacci heap. As before, at most n blocking flows have to be computed, because the distance from s

to t in the level graph increases by at least one each time. This gives an overall worst-case time bound of $O(n^3 \log n)$ with a heap-based priority queue or $O(n^3)$ with a Fibonacci heap.

References

[1] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *Regional Conference Series in Applied Mathematics*. SIAM, 1983.

[2] E. A. Dinic. Algorithm for solution of a problem of maximal flow in a network with power estimation. *Soviet Math. Doklady*, 11:1277–1280, 1970.

[3] V. M. Malhotra, M. Pramodh-Kumar, and S. N. Maheshwari. An $O(V^3)$ algorithm for finding maximum flows in networks. *Information Processing Letters*, 7:277–278, 1978.