Lecture 18 Still More on Max Flow

18.1 Dinic's Algorithm

We follow Tarjan's presentation [100]. 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 in 1970 [29]. 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*.

The following algorithm describes one 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. 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 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.

Algorithm 18.1 (Dinic [29])

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 $O(mn^2)$.

18.2 The MPM Algorithm

The following algorithm given by Malhotra, Pramodh-Kumar, and Maheshwari in 1978 [77] produces a max flow in $O(n^3)$ time. The overall structure is

similar to the Edmonds-Karp or Dinic algorithms. Blocking flows are found for level graphs of increasing depth. The algorithm's superior time bound is due a faster ($O(n^2)$) 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. Intuitively, the capacity of a vertex is the maximum amount of commodity that can be pushed through that vertex.

Definition 18.2 The *capacity* $c(v)$ of a vertex v is the minimum of the total capacity of its incoming edges and the total capacity of its outgoing edges:

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

□

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 is computed. If t does not appear in L, 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 according to Definition 18.2. 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 back to 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 Fibonacci heaps, it takes $O(n \log n)$ time amortized 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.

Note that when we delete edges, we must decrement the capacities of neighboring vertices; this is done using the decrement facility of Fibonacci heaps.

The loop thus achieves a blocking flow in $O(n^2)$ time. 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 $O(n^3)$.

The max flow problem is still an active topic of research. Although $O(n^3)$

remains the best known time bound for general graphs, new approaches to the max flow problem and better time bounds for sparse graphs have appeared more recently [38, 98, 4, 41, 95, 37].

18.3 Applications of Max Flow

Bipartite Matching

Definition 18.3 A *matching* M of a graph G is a subset of edges such that no two edges in M share a vertex. We denote the size of M by $|M|$. A *maximum matching* is one of maximum size. □

We can use any max flow algorithm to produce a maximum matching in a bipartite graph $G = (U, V, E)$ as follows. Add a new source vertex s and a new sink vertex t, connect s to every vertex in U, and connect every vertex in V to t. Assign every edge capacity 1. The edges from U to V used by a maximum integral flow give a maximum matching.

Minimum Connectivity

Let $G = (V, E)$ be a connected undirected graph. What is the least number of edges we need to remove in order to disconnect G? This is known as the *minimum connectivity problem*.

The minimum connectivity problem can be solved by solving $n - 1$ max flow problems. Replace each undirected edge with two directed edges, one in each direction. Assign capacity 1 to each edge. Let s be a fixed vertex in V and let t range over all other vertices. Find the max flow for each value of t, and take the minimum over all choices of t. This also gives a minimum cut, which gives a solution to the minimum connectivity problem.