

Max Flow: An $O(m^2n)$ Algorithm

Theorem 1. *If the augmenting path in the Ford-Fulkerson algorithm is found by a breadth-first search (i.e. the algorithm uses a path of minimum number of edges), the run time of the algorithm is $O(m^2n)$, where m is the number of edges and n is the number of vertices.*

Level Graph

Definition 1. *The level graph L_G of G is the directed breath-first graph of G with root s with sideways and back edges **deleted**. The **level** of a vertex u is the length of the shortest path from s to u in G .*

Observations:

1. The level graph has no edges from level i to level j for $j \geq i + 2$.
2. Any shortest path from s to any other vertex is a path in the level graph.
3. Any path with either a back or a sideways edge of the level graph will be strictly longer.

Lemma 1. (a) Let p be an augmenting path of minimum length in G . Let G' be the residual graph obtained by augmenting along p and let q be an augmenting path of minimum length in G' . Then $|q| \geq |p|$. Thus the length of the shortest augmenting paths cannot decrease by applying the above heuristic.

(b) We can augment along shortest paths of the same length at most $m = |E|$ times before the length of the shortest augmenting path must increase strictly.

Proof

p is some path in the level graph L_G .

After augmenting along p , one edge will disappear and at most $|p|$ new edges will appear in the residual graph G' .

These new edges are back edges and cannot contribute to a shortest path from s to t as long as t is reachable from s in the level graph.

Every time we augment using a path in L_G we lose an edge in L_G . Thus we can augment at most m times, before t is no longer reachable in L_G . Then any augmenting path must use a back or side edge and must be strictly longer.

Edmonds-Karp Algorithm

1. Find the level graph L_G .
2. Repeatedly augment along paths in L_G , updating residual capacities and deleting **saturated** edges (zero capacity edges) until t is no longer reachable from s .
3. Calculate a new level graph from the residual graph at that point and repeat.
4. Continue as long as t is reachable from s (in the residual graph).

Running time

After each level graph calculation, the distance between s and t increases by 1.

Thus, there are at most n level graph calculations.

$O(m)$ time to calculate a level graph.

The total number of augmentations is at most $O(mn)$ and each augmentation takes $O(m)$. Thus total time is $O(m^2n)$.

An $O(mn^2)$ Algorithm

Similar to Edmonds-Karp, but we find augmenting paths in the level graph L_G more carefully.

Blocking flow: The flow when there is no path from s to t in L_G .

Instead of constructing a blocking flow path by path, we construct a blocking flow all at once by finding a maximal set of minimum-length augmenting paths. Each such construction is a phase.

In each phase we will construct augmenting paths in L_G in a depth-first fashion.

Dinic's Algorithm: One Phase

1. Initialize: Construct a new level graph L_G .

Let $u = s$ and $p = s$. (u will denote the vertex currently being visited and p is a path from s to u .)

Go to **Advance**.

2. Advance: If there is no edge out of u , go to **Retreat**. Otherwise, let (u, v) be such an edge.

Set $p = p.[v]$ and $u = v$.

If $v \neq t$ then go to **Advance** else go to **Augment**.

3. Retreat: If $u = s$ then **Stop**.

Otherwise, delete u and all adjacent edges from L_G and remove u from the end of p .

Set $u =$ last vertex on p . Go to **Advance**.

4. Augment: Let Δ be the bottleneck capacity along p .

Augment path flow along p and adjust 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 .

Set $p =$ the portion of p up to and including u . Go to **Advance**.

Running time

1. **Initialize:** $O(m)$ time.
2. **Advance:** $O(mn)$ time.

There are at most $2mn$ advances in each phase, because there at most n advances before an augment or retreat and there are at most m augments and n retreats.

3. **Retreat:** $O(m + n)$ time.

At most n retreats in each phase. Each retreat takes $O(1)$ time plus time to delete edges which is $O(m)$.

4. **Augment:** $O(mn)$ time.

At most m augments in each phase and each augment takes $O(n)$ time. Thus each phase requires $O(mn)$ time and there are at most n phases.

Hence total running time is $O(mn^2)$.

An $O(n^3)$ Algorithm

Similar to Edmonds-Karp and Dinic, but blocking flows are found for level graphs in $O(n^2)$ time.

We consider the capacity of a vertex as opposed to the capacity of an edge.

Definition 2. *The capacity $c(v)$ of a vertex v in a flow network is the minimum of the total capacity of its incoming edges and the total capacity of its outgoing edges:*

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

The algorithm proceeds in phases.

In each phase, we compute the residual graph for the current flow and also compute the level graph.

If t does not appear we are done. Otherwise, all vertices not on a path from s to t in the level graph are deleted. We now find a blocking flow as follows.

MPM Algorithm: One Phase

Step 1: Find a vertex v of minimum capacity d . If $d = 0$ goto step 2 else Goto **Step 3**.

Step 2: Delete v and all its incident edges and update the capacities of the neighboring vertices. Goto **Step 1**.

Step 3: Push d units of flow from v to sink and **pull** d units of flow from the source to v to increase the flow through v by d :

Push to sink: Saturate the outgoing edges of v in order, leaving at most one partially saturated edge.

Repeat this process on each vertex (all the way up to t) that received flow during the saturation of the edges out of v .

Delete all saturated edges.

Pull from source: Saturate the incoming edges of v in order, leaving at most one partially saturated edge.

Repeat this process on each vertex (all the way back to s) from which flow was taken during the saturation of the edges out of v .

Delete all saturated edges.

Delete v and all its remaining incident edges from the level graph. Update the capacities of the neighbors. Goto **Step 1**.

Running time

Time needed for one phase:

$O(m)$ time to compute the residual graph and level graph using BFS.

$O(n \log n)$ amortized time to find and delete a vertex of minimum capacity using Fibonacci heaps.

$O(m)$ time to delete all the (fully) saturated edges.

$O(n^2)$ time to do partial saturations, because it is done at most once in step 3 at each vertex for each choice of v in step 1.

$O(n^2)$ time to decrement capacities of incident vertices using Fibonacci heaps.

Thus, one phase takes $O(n^2)$ time and there are at most n phases.