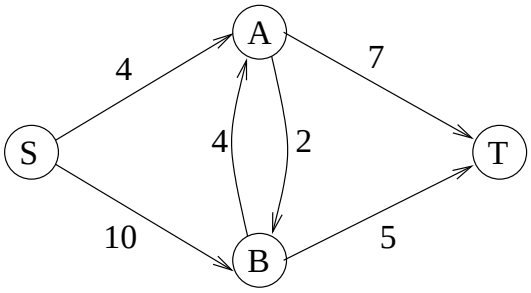


Figure 27: Each edge in this graph is labeled with its capacity.

8 Introduction to Network Flow

8.1 Flow-Related Problems

The optimization problems we have examined so far have had no known algorithms that solved the problems exactly, requiring us to devise approximation algorithms. We will now examine a class of problems that deal with flow in networks, for which there exists efficient exact algorithms. One can view these flow-related problems as tools. As we will see later, many problems can either be reduced directly to flow, or can be approximated by solving an appropriately constructed flow problem.

Maximum flow problem: Given a directed graph $G = (V, E)$, we assign to each edge $uv \in E$ a nonnegative (and integral in our discussion) *capacity* $cap(uv)$. If $uv \notin E$ we assume $cap(uv) = 0$. One of the nodes of G is designated as the source s, and another as sink t. The goal of the problem is to maximize the flow from s to t, while not exceeding the capacity of any edge.

Note: Although the *max-flow* problem is defined in terms of directed graphs, almost all techniques we will discuss apply to undirected graphs as well.

Maximum flow problems arise in a variety of contexts. For example:

1. *Networks*

   Every edge represents a communication link in the network. Information has to travel from s to t. Capacity of an edge edge is equal to its bandwidt, i.e. number of bits per second that can travel along this edge.

2. *Transportation*

   A truck is loaded at s and has to deliver its cargo to t. The edges of the graph are roads, and the capacities are the number of trucks per hour that can travel on that road.

3. *Bridges*

   Each vertex represents a physical location and each edge represents a bridge between two locations. The capacity of an edge is the cost to block the bridge it represents. The goal is to disconnect s and t while incurring the minimum cost in blocked bridges.

There are two sides to our flow optimization problem. We will state them informally first.

1. Transport maximum rate of material from s to t, as in examples 1 and 2 above. This is known as the *max-flow* problem. For example, as in the graph in Figure 27, we can construct the following flow:

   - Send 4 units along the path $s \to a \to t$.
   - Send 3 units along the path $s \to b \to a \to t$.
   - Send 5 units along the path $s \to b \to t$.

   The total flow is 12, which happens to be the maximum flow for this graph.

2. Cut edges of G to disconnect s and t, minimizing the cost of the edges cut, as in example 3 above. This is known as the *min-cut* problem.

   In the graph in Figure 27, it is easy to see that cutting edges at and bt will disconnect s and t, since there will be no edges going into t. The capacity of these edges is $cap(at) + cap(bt) = 12$. This happens to be the lowest capacity cut that separates s and t.

We will show the formal equivalence of *max-flow* and *min-cut* problems later.

We will now define the *max-flow* problem formally. For notational convenience, consider "mirror" edges. For each edge $e = (u, v)$ in $G = (V, E)$ with capacity $cap(e)$ and flow $f(e)$, we will add a "mirror" edge $e' = (v, u)$ with capacity 0 and flow $-f(e)$. Note that if both uv and vu are present in E, we will now have four edges between u and v.

Definition 8.1. *A* flow *is a real-valued function f defined on the set of edges $uv \in V \times V$ of the graph $G = (V, E)$ subject to two constraints:*

1. Capacity constraint. *$f(uv) \leq cap(uv) \forall u, v \in V$, where $cap(uv)$ is the capacity of edge uv.*
2. Conservation constraint. *The following equation holds for all $u \in V - \{s, t\}$*

$$\sum_{v:uv,vu \in E} f(uv) = 0.$$

The capacity constraint limits the flow from above. Note that the "mirror" edges added to the graph satisfy this constraint if the flows through edges of E are nonnegative.

The conservation constraint forces inflow and outflow for a node to sum to 0 for all nodes, where inflow is measured in negative numbers and outflow is measured in positive numbers. This requirement does not hold for the source (s) or the sink (t), but together their flow sums up to 0. (Can you explain why ?)

We can see in Figure 28 that all the capacity constraints and conservation constraints are satisfied. The shown flow is therefore a *feasible flow* on G.

Definition 8.2. *A* cut *in a graph $G = (V, E)$ is a partition of the set of vertices of the graph, V, into two sets A and $V - A$, where $A \subseteq V$.*

We can also think of a cut as a set of edges that go from A to $V - A$, i.e. all edges uv such that $u \in A, v \in V - A$. If we remove (cut) these edges from the graph, no vertex in A will be connected to a vertex in $V - A$. The capacity of a cut is defined as follows:

$$cap(A, V - A) = \sum_{u \in A, v \in V - A, uv \in E} cap(uv).$$

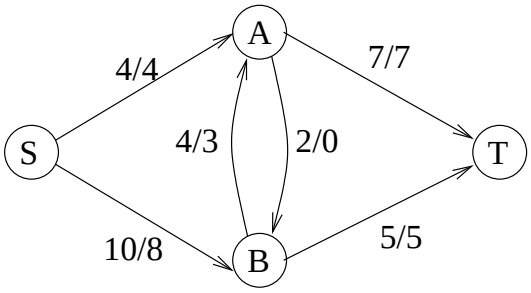


Figure 28: A feasible flow in a graph. Each edge is labeled with capacity/flow. Mirror edges are not shown.

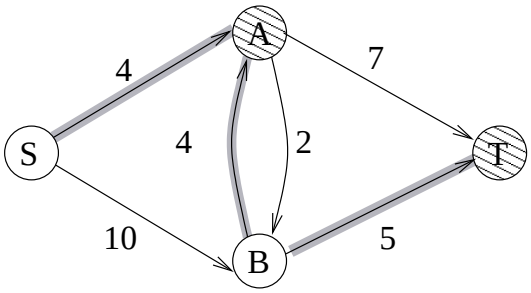


Figure 29: An example of a cut in a graph. Let the set A consist of the unshaded nodes and $V - A$ consist of the shaded nodes. Edges between the partitions of the cut $(A, V - A)$ are highlighted. The capacity of the cut is $cap(sa) + cap(ba) + cap(bt) = 12$.

The direction of edges is important in the definition of a cut. Capacities of edges that go from $V - A$ to A are not counted towards the capacity of the cut (see Figure 29). This is because we want the capacity of a cut to limit the flow going through that cut in one direction.

Definition 8.3. *The* flow across a cut $(A, V - A)$ *on a graph* $G = (V, E)$ *is given by:*

$$f(A, V - A) = \sum_{u \in A, v \in V - A} f(uv).$$

The flow across a cut is a number, not a function. Note the difference between this definition and the definition of the capacity of a cut. To calculate a net flow through a cut we must take into account flows going in both directions. Hence, the condition $uv \in E$ is removed allowing us to consider mirror edges from A to $V - A$.

Definition 8.4. *An s-t cut on a graph* $G = (V, E)$ *is a cut* $(A, V - A)$ *on* G *such that* $s \in A$ *and* $t \in V - A$.

Maximizing Flow. In order to maximize flow, it is necessary to maximize $f(\{s\}, V - \{s\})$, which is the sum of the flows on all edges going out of the source. Equivalently, we can maximize $f(V - \{t\}, \{t\})$, which is the sum of the flows on all edges going into the sink. It can be proven that these values are equal. In fact, it can be proven that the flows across all s-t cuts are the same (see homework). Denote this value by $|f|$.

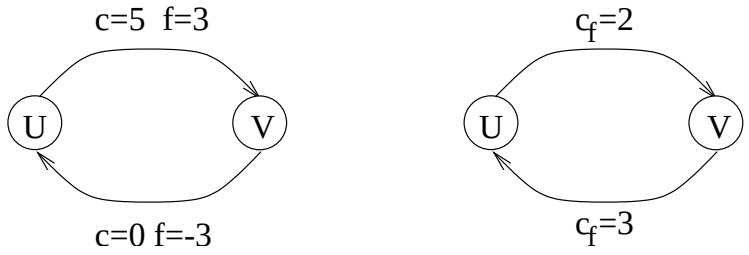


Figure 30: Residual capacities of edges. We can see that if a flow of 3 has been pushed across an edge with capacity 5, we could push 2 more units of flow or 3 fewer units of flow across that edge. This gives us our two residual edges of 2 and 3 respectively.

8.2 Residual Graphs

Currently we have to keep track of two values for each edge of G, capacity and flow. *Residual graphs* will allow us to only keep track of capacity for each edge, updating this capacity as the flow changes. Residual graphs represent how much additional flow can be "pushed" along a given edge. We can always construct the current flow in G from the current residual graph and original capacities of G.

To create a residual graph, consider for each edge uv the remaining capacity (residual capacity) of that edge after some flow $f(uv)$ was pushed across it. Denote this value as $cap_f(uv)$, then

$$cap_f(uv) = cap(uv) - f(uv).$$

Figure 30 shows residual capacities of an edge uv of G and its "mirror" edge vu after a flow of 3 units was pushed across uv.

Definition 8.5. *A residual graph of $G = (V, E)$ given a flow f is defined as $G_f = (V, E_f)$, where*

$$E_f = \{uv \in VxV \mid cap_f(uv) > 0\}.$$

If G_f has a directed path from the source to the sink, then it is possible to increase the flow along that path (since by the above definition each edge in E_f has positive capacity). Increasing the flow on such a path is called *augmentation*.

Consider the following simple algorithm for finding the *max-flow* in a graph.

Algorithm for finding *max-flow* in a directed graph

1. `Compute initial residual graph (assume flow on each edge is 0).`
2. `Find a directed path from the source to the sink.`
3. `Augment the flow on the path by amount equal to the minimum residual capacity along this path.`
4. `Update the residual graph.`
5. `Repeat from step 2 until there is no augmenting path`

We need to show that the algorithm will eventually terminate. Observe that if the capacities are integer, we will always augment the flow by at least one unit. Thus, there is an upper bound on the

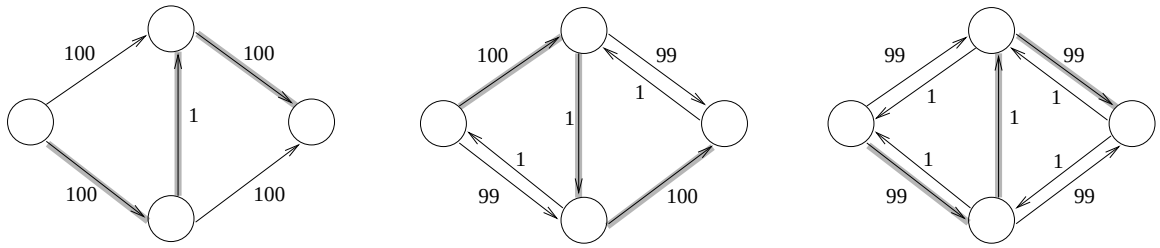


Figure 31: This figure shows a sequence of augmentations. Although we can see that the value of the *max-flow* is 200, the algorithm may make unlucky depth-first choices and perform 200 iterations, the first two of which are shown here.

number of iterations of the algorithm. Indeed, if the maximum capacity of an edge is U and number of nodes is n, then the algorithm will perform at most Un iterations. This is because the upper bound on the flow in the graph is $cap(\{s\}, V - \{s\})$, which is the maximum amount of flow that can go out of s.

The algorithm used at step 2 runs in time $O(m)$ (think about depth-first graph traversal), and the total running time of the algorithm is therefore $O(nmU)$. This is not a polynomial algorithm, since the running time is a function of U. However, if U is small, the algorithm might be efficient. In particular, for constant U we have $O(nm)$ running time.

The following example illustrates that the above algorithm is, in fact, non-polynomial. In other words, it shows that the algorithm is indeed slow and that our analysys is not too pessimistic. Figure 31 shows an example of execution that terminates in $\Omega(Un)$ iterations.

So now we know that, in general, our algorithm is slow. But note that we have not even shown that the algorithm produces an optimum solution ! We will prove this fact in the next section. Meanwhile we will note several easy to see properties:

1. The algorithm builds a flow which cannot be augmented, i.e. is maximal. (We have not shown yet that it is *maximum*.)
2. If the *max-flow* problem has integer capacities, then the algorithm builds an integer flow.

8.3 Equivalence of min cut and max flow

In this section we will use the following terms:

max-flow—the maximum flow that we can pump from the source s to the sink t (the flow which maximizes $|f|$, where $|f|$ is the flow across any s-t cut),

min-cut—the s-t cut of the smallest capacity.

Theorem 8.6. *Given a capacitated graph $G = (V, E)$, the following statements are equivalent:*

a. f is a max flow.

b. There is an s-t cut such that the capacity of the cut is equal to the value of f.

c. There is no augmenting path in E_f (the residual graph with respect to the flow f).

Proof:

b ⇒ a Notice that for every s-t cut $(A, V - A)$, the flow $f(A, V - A)$ from A to $V - A$ is less than or equal to $cap(A, V - A)$, the capacity of the cut:

$$f(A, V - A) = \sum_{u \in A, v \notin A} f(uv) \leq \sum_{u \in A, v \notin A} cap(uv) = cap(A, V - A).$$

Therefore, the flow across any cut cannot exceed its capacity.

Also notice that $f(A, V - A)$ is the same for all s-t cuts $(A, V - A)$. (This is established in homework.) Consequently, the amount of any flow is less than or equal to the capacity of the minimum s-t cut. So if $|f|$ equals the capacity of the min cut, it cannot be increased any further, and therefore, f is a max flow.

a ⇒ c Assume to the contrary that f is a max flow and that there exist at least one augmenting path. But then we can use the augmenting path to augment f, increasing its value and contradicting the assumption that f is max flow. Thus, if f is max flow then no augmenting path can exist.

c ⇒ b This is probably the most "interesting" direction. First, define the set $A = \{v \in V \mid v \text{ is reachable from } s \text{ in } E_f\}$. Recall the definition of E_f. If a vertex v is in A, it means that there is a path of edges with positive residual capacity from s to v.

The sink t is not in A since there is no augmenting path. The vertex s is in A trivially. Now, consider an s-t cut $(A, V - A)$. For every $uv \in E$, where u in A and v in $V - A$, we have $f(uv) = cap(uv)$, for if this were not the case, v would be reachable from s. Similarly, for any $vu \in E$ where u in A and v in $V - A$, we have $f(uv) = 0$. Summing over the edges, we obtain $f(A, V - A) = cap(A, V - A)$.

∎

Observe that our *max-flow* algorithm from the previous section creates a flow such that there are no remaining augmenting paths, i.e. exhibits the properties of part c of our theorem. Thus, our algorithm finds a maximum flow. Observe that if the capacities are integer, we will always augment by an integer amount (easy proof by induction on the number of augmentations). Thus, we have the following useful corollary:

Corollary 8.7. *If all capacities are integers, then there is an all-integer max flow. This flow can be built by iteratively augmenting paths in the residual graph.*

This corollary does not mean that there are no fractional solutions. They usually do exist, but there is at least one integral solution.

8.4 Polynomial max-flow algorithms

8.4.1 Fat-path algorithm and flow decomposition

In this section we present a polynomial time algorithm for solving the max flow problem. Similarly to the Ford-Fulkerson algorithm presented earlier, this algorithm is based on the idea of successive augmentations. However, the previous algorithm had a running time exponential in the length of the representation of the input. The main difference here is that, at each step, we try to find a "fat" augmenting path, i.e. we try to augment by as much as possible. The running time of the algorithm is $O((m + n \log n)m \log(nU))$, where n is the number of nodes, m is the number of edges and U is the maximum edge capacity. If we represent U in our input in binary, this algorithm has a polynomial running time, though not strongly polynomial.

The algorithm The algorithm itself is very simple and relies on our ability to always find an augmenting path of maximum residual capacity, i.e. a path from s to t such that its bottleneck residual capacity is greatest. This can be done efficiently (in $m + n \ln n$ time) by a modified Dijkstra's algorithm.

The algorithm can be defined as follows:

1. Initially f is 0. Find an augmenting path of maximum residual capacity.
2. Augment the flow along this path by the path's bottleneck capacity.
3. Recompute G_f. Repeat steps **1** and **2** until there are no augmenting paths remaining.

First, notice that this algorithm is in fact correct since it stops only when it can no longer find an augmenting path, which by max-flow/min-cut theorem implies that we have a max flow.

Analysis The proof of the running time of this algorithm will rely on two claims. The first is the Decomposition Theorem, and the other is that the optimal flow minus current (feasible) flow is a feasible flow in the current residual graph.

The main idea behind the Decomposition Theorem is that it is possible to separate every flow into a flow along a limited number of cycles, C_i, and along paths from source to sink, P_i. In order to simplify notation, we will use C_i to denote a vector where each coordinate corresponds to an edge and where it is equal to 1 if and only if the corresponding edge belongs to cycle C_i. We will use similar notation for paths. Formally, the *Decomposition Theorem* can be stated as follows:

Theorem 8.8. *The flow in a graph G can be subdivided into flows along cycles C_i and paths P_i in such a way that:*

$$f = \sum_i f(C_i) \cdot C_i + \sum_i f(P_i) \cdot P_i$$

where f is the flow vector and $f(C_i), f(P_i)$ are scalars, representing flows assigned to paths and cycles. Moreover, the total number of paths and cycles is bounded by m, the number of edges in G.

Proof: We can prove the Decomposition Theorem by construction. Let G^f denote the flow graph, i.e. it has an edge uv if $f(uv) > 0$. We will use the following procedure for decomposing the flow. The idea is to iteratively find paths and cycles, assign flow to them, and update the flow graph G^f by removing this flow from the appropriate edges. At each step, we maintain that the current flow in G^f together

with the flows in already constructed paths and cycles sums up to exactly the original flow. At the end, we have G^f flow equal to zero.

1. Start at the source s and follow edges in G^f until either we reach the sink or we close a cycle. Conservation constraints imply that if, during our walk, we arrived to some node $v \notin \{s, t\}$ over edge uv with $f(uv) > 0$, there has to be another edge vw with $f(vw) > 0$. In other words, we will not "get stuck". There are 2 cases to consider:
   (a) We reached t over a simple path. In this case, denote this path by P, compute the minimum of all the flow values on edges of this path ($\min_{vw \in P} f(vw)$) and set $f(P)$ to this value. Update the flow f by setting:
   $$f \leftarrow f - f(P) \cdot P$$
   (b) We visit a node for the second time. This means that our path includes a cycle. Denote this cycle by C. Compute the minimum of all the flow values on edges of this cycle ($\min_{vw \in C} f(vw)$) and set $f(C)$ to this value. Update the flow f by setting:
   $$f \leftarrow f - f(C) \cdot C$$
2. If there is at least one edge from s with non-zero flow on it, go back to (1).
3. If there are no outgoing edges from s in current G^f, repeat the same with t.
4. When we reach this point, s and t do not have outgoing edges in G^f. At this point we have a flow f that *satisfies conservation constraints at all nodes*, including s and t. (Try to formally prove this !)
5. Now repeat the following, until there are no more edges in G^f:
   (a) Pick an edge uv in G^f, i.e. $f(uv) > 0$. By conservation constraints, there has to be an edge vw with $f(vw) > 0$. Move to vw and continue. This walk can stop only if we close a cycle. Denote this cycle by C.
   (b) Compute the minimum of all the flow values on edges of C ($\min_{vw \in C} f(vw)$) and set $f(C)$ to this value. Update the flow f by setting:
   $$f \leftarrow f - f(C) \cdot C$$

Observe that each time we find a path or a cycle, we update flow f in a way that guarantees that at least one of the edges leaves G^f. Thus, the final decomposition will include at most m cycles and paths. ∎

We will need the following lemma for the analysis of the running time:n

Lemma 8.9. *If f^* is the maximum flow on a graph G, then for any feasible flow f, $f^* - f$ is also a feasible flow on the residual graph G_f.*

Define $cap(uv)$ to be the capacity of the edge (u, v) in G, and let $cap_f(uv)$ be the capacity of an edge in the residual graph G_f.

For any edge (u, v) of G_f, we have that if $f(uv) \leq f^*(uv)$, then

$$f^*(uv) \leq cap(uv) \Rightarrow f^*(uv) - f(uv) \leq cap(uv) - f(uv) = cap_f(uv).$$

Thus, the flow of the edge (u, v) in the flow $f^* - f$ does not exceed the capacity of the residual graph G_f. Similar proof shows that the claim is true for the case where $f(uv) > f^*(uv)$. ∎

Now we are ready to prove the bound on the running time of the algorithm. This proof will show basically that at each augmentation cycle, the algorithm will augment by at least $1/m$ of the remaining difference between the value of maximum flow and the value of the current flow. After m iterations, the current (before the first iteration) flow will be reduced by a factor of approximately $1/e$. Using this, one can show that the actual number of augmentation cycles that can occur is bounded from above by $O(m \log(nU))$ where U is the maximum capacity of any edge in the graph.

Theorem 8.10. *The fattest-augmentation path algorithm terminates in $O(m \log(nU))$ iterations.*

Consider the flow f which we have after some number of augmentation steps. If f^* is the optimal flow, then from our lemma we know that $f^* - f$ is a feasible flow. Then, from the Decomposition Theorem, we can break $f^* - f$ into at most m paths and cycles.

Observe that $|f^* - f| = |f^*| - |f|$. (Can you formally prove this ?) Moreover, since cycles do not contribute to the flow value, this means that the flow along the paths in the decomposition sums to exactly $|f^*| - |f|$. Thus, there exists at least one path (denote it by P) in the decomposition of flow $f^* - f$ which has a flow of at least $(1/m) * (|f^*| - |f|)$.

Observe that, since $f^* - f$ is a feasible flow in the current G_f, we have $\forall uv \in P, c_f(uv) \geq f(P) \geq (1/m) * (|f^*| - |f|)$. Thus, our algorithm will find a path (not necessarily P) whose bottleneck capacity is not less than $(1/m) * (|f^*| - |f|)$.

Let F^i denote the total flow from source to sink at iteration i of the algorithm, we must have

$$F^i \geq F^{i-1} + \frac{F^* - F^{i-1}}{m}$$

Allowing δ_i to denote $F^* - F^i$ (where, since there is no flow initially, $\delta_0 = F^*$), we have the following inequality:

$$\delta_i = F^* - F^i \leq F^* - (F^{i-1} + \frac{F^* - F^{i-1}}{m}) = \delta_{i-1} - \frac{\delta_{i-1}}{m}$$

Consequently,

$$\delta_i \ \leq \delta_0 \ (1 - \frac{1}{m})^i = F^*(1 - \frac{1}{m})^i$$

Since we are dealing with integer capacities, all flows will also be integers, so if we are within 1 of the solution, we are done. Formally, if $\delta_k \leq F^*(1 - \frac{1}{m})^k < 1$, then $F^k = F^*$. Taking a natural logarithm of both sides of the inequality, we obtain:

$$0 > \ln F^* + k \ln (1 - \frac{1}{m})$$

Using the Taylor expansion $\ln (1 - x) \approx -x - x^2/2 + O(x^3) < -x$, we find that any $k > m \ln F^*$ satisfies this equation, so at most $m \ln F^*$ iterations are needed to get to a max flow state.

Because at most n edges can emanate from s, we conclude that if U denotes the maximum capacity of any edge in the graph, F^* is bounded from above by nU since this is the most that can possibly flow out of the source. ∎

If we remember that finding a maximum augmenting path using Dijkstra's algorithm takes $O(m + n \log n)$ time, we obtain the final running time of $O((m + n \log n)m \ln nU)$. Observe that this is bounded from above by $O(m^2 \log nU \log n)$.

8.4.2 Polynomial algorithm for Max-flow using scaling

We first explore means to extend max-flow solutions of graphs with approximate integer capacities, to find the max-flow in the original graph. Let us consider the case when the approximate capacities are represented by the first i significant bits of the bit-representation of the capacities, denoted by $\text{cap}_i(\text{e})$ for an edge e.

The algorithm proceeds in phases, where phase i computes max-flow f_i for graph with capacities cap_i. Max-flow for $i = 0$ is trivially 0.

The ith significant bit can be either 1 or 0 and hence $\text{cap}_i(\text{e})$ is either $2{*}\text{cap}_{i-1}(\text{e})$ or $2{*}\text{cap}_{i-1}(\text{e})+1$. Given f_{i-1}, we need to quickly convert it into f_i. This can be viewed as two steps: first convert it into max flow that satisfies capacities $2{*}\text{cap}_{i-1}$, and then increment some of these capacities by 1 to get cap_i, and update the flow appropriately.

Note that if we have max-flow for certain capacities, then doubling (coordinate-wise) this flow gives us max-flow for doubled capacities. This can be verified by observing that after doubling both flow and capacities, the min-cut is remains saturated and the conservation and capacity constraints are satisfied as well. Thus, $2f_{i-1}$ is max flow for capacities $2{*}\text{cap}_{i-1}$.

Given max-flow for some given set of capacities, consider what will happen if one increments some of the capacities by 1. Observe that the residual capacity of the min-cut grows by at most the number of edges in the min-cut since the residual capacity initially was 0. Thus, incrementing capacities by at most 1 each can increase the value of max-flow by at most m total. This, in turn, implies that, given max-flow before capacity increment, we only need at most m augmentations to get the max-flow for the graph with incremented capacities.

Applying this reasoning to our context implies that to compute f_i, we first double f_{i-1}, compute residual graph with respect to cap_i, and augment at most m times.

There are at most $\log U$ phases, each phase consisting of at most m augmentations, where each augmentation takes at most $O(m)$ time (e.g., using DFS). Total running time is $O(m^2 \log U)$, which is polynomial in the *size of the input*. This type of approach is usually called *scaling*.

8.4.3 Strongly polynomial algorithm for Max-flow

Although our previous algorithm was polynomial, we are not satisfied with the $\log U$ term that appears in our running time, where U is the maximum edge capacity. The reason is that even though our algorithm is polynomial in the *size of the input*, its running time depends on the precision of the capacities. Observe that if all capacities are multiples of some number, we can divide by this number without changing the problem. Moreover, the data representation for the capacities itself may be different and involve more than $\log U$ bits. In this section we will describe an algorithm with running time that depends only on the size of the input graph, i.e. n and m. Such algorithms are called "strongly polynomial". (Strictly speaking, there are several other formal requirements. In particular, we have to make sure that the number of bits needed to represent intermediate results is bounded by a polynomial in n and m times the number of bits in the input.)

The basic idea behind this algorithm is to augment along the shorter paths in the graph first. We begin by constructing a *layered network* for the residual graph G_f, where each layer k consists of nodes that can be reached in k "hops" from the source. We also ensure that nodes in layer k cannot be reached in less than k hops from the source. We do this by using Breadth First Search on the current residual graph G_f. Specifically, we start at the source and place all nodes that can be reached in one hop from

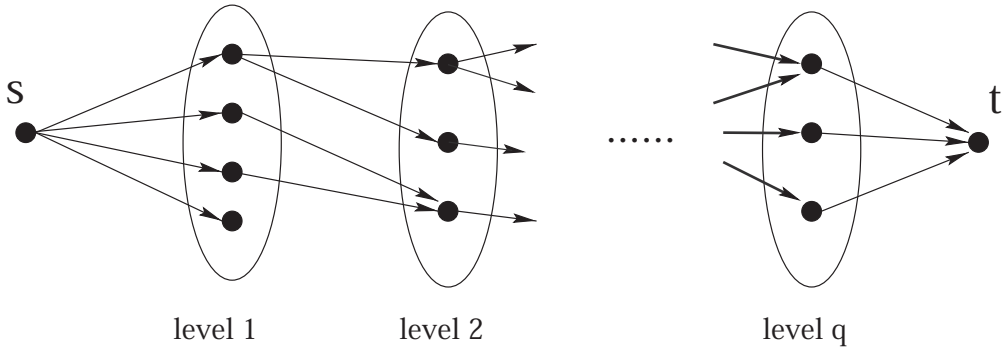


Figure 32: Layered Network

it into layer 1; then we take all untouched nodes that can be reached from layer 1 nodes and place them into layer 2. We continue with the same approach iteratively until we reach the sink or have traversed all edges in G. The running time of this algorithm is the running time of BFS, namely, $O(m)$. Our layered network may look as shown in Figure 32.

We note the following useful facts about this layered network:

1. There can be no edges that skip a layer forward, since if such an edge existed, its endpoint would have to be placed in the level just one greater than its rear endpoint.
2. There can be at most n levels in the network, since no node could be located in 2 different levels, and G contains n nodes.
3. If the sink cannot be reached after n levels, there are no available augmenting paths from source to sink, so we already have max-flow and are done.
4. Nodes may exist beyond the sink, which do not belong to any layer. These nodes will not participate in the current phase since they do not lie on shortest paths.

Our algorithm shall proceed as follows:

1. Construct a layered network from G_f. If it's not possible to reach the sink t in an n-layer network, we have max-flow by the previous remarks and we are done.
2. Find any forward augmenting path in the n-layer network and augment along this path. Only forward residual edges are allowed.
3. Update the residual graph G_f.
4. Repeat step 2 until there is no forward augmenting path left and then restart from step 1.

The above algorithm operates in phases, where at each phase we build a new layered network and perform several augmentations. Observe that since we restrict our search for augmenting paths to those paths that have only forward edges (i.e. edges from layer i to layer $i+1$), every augmentation reduces the number of forward edges by at least 1. This is because the augmentation will saturate the forward edge with the smallest capacity along the augmenting path, essentially removing this edge from G_f. This introduces a "reverse" residual edge, but such edges are not allowed for augmentation until the next phase, i.e. until we rebuild the layered network.

Note that all augmentations during a single phase are along same length paths, where this length is equal to the distance between s and t in the residual graph at the beginning of the phase (i.e., number of layers in the layered network). A phase is terminated when there are no more augmenting paths from using only forward edges. Note that a path using a backwards edge has to be longer than the number of layers in this phase. Thus, if at the beginning of a phase the s to t distance was k, then at the end of the phase it will be at least $k + 1$. In other words, we will have more layers in the next phase.

The running time of the algorithm can be computed as follows:

1. The number of phases is bounded by n because there can be at most n levels in the network and each subsequent phase starts with more layers in the layered network.
2. The number of augmentations per phase is bounded by m because after each augmentation the number of forward edges in the layered network is reduced by at least 1.
3. We can find an augmenting path in $O(m)$ by running DFS in the layered network, disregarding all but the forward edges in G_f.

Therefore, the total running time is $O(m^2 n)$.

The above analysis is not tight. We can significantly improve the bound by reducing the time wasted while searching for augmenting paths. Consider an edge traversed by DFS. If we backtrack along this edge during this DFS, we consider this as a waste. The main observation is that if, during a single phase, we backtrack along an edge, then it is useless to consider this edge again until the next phase. The reasoning is as follows: if we backtrack along the edge uv then there is no forward path from v to t. But augmentations during a phase can only introduce back edges, and hence once we notice that there is no forward augmenting path from v to t, then such path will not appear until the next phase. It is important to note that, initially, there might be a forward path from v to t that is "destroyed" during the phase.

The above discussion implies that each time we backtrack along an edge during DFS, we can mark this edge as "useless" (essentially deleting it) *until the next phase*. Thus, we backtrack over each edge at most once during the phase, which gives us $O(m)$ bound on "wasted" work during the phase.

The "useful" work consists of traversing edges forward, without backtracking. Notice that there can be at most n such edges during a single DFS. (In fact, the number is bounded by the number of layers in the current phase network.)

Combining the above results, we see that the total amount of wasted work during a phase is bounded by $O(m)$ and the total amount of "useful" work is bounded by $O(mn)$ i.e $O(n)$ per augmentation. [Why can't we claim $O(n)$ augmentations?] Since building a layered network takes $O(m)$ time, we get $O(mn^2)$ bound on the running time of the algorithm. Notice that this is a significant improvement over the $O(m^2 n)$ time computed earlier, since m can be as large as $\Theta(n^2)$.

Comparing this bound to the $O(m^2 \log U)$ bound we got in the previous section, we see that our new bound is not always better. The best running time for a strongly polynomial max-flow algorithm is $O(nm \log(n^2/m))$, which we will discuss in subsequent sections.

8.5 The Push/Relabel Algorithm for Max-Flow

8.5.1 Motivation and Overview

In the previous lectures we introduced several max-flow algorithms. There are cases though, where those algorithms do not have good performance, since the only tool we have been using up to now is augmentation. Consider a graph with a many-hop, high-capacity path from the source, s, to another node b, where b is then connected to the sink, t, by many low-capacity paths. Figure 33 illustrates such a scenario. In this topology, the max-flow algorithms introduced earlier must send single units of flow individually over the sequential part of the path (from s to b), since no full path from s to t has a capacity of more than one. This is clearly a fairly time-consuming operation, and we can see that the algorithm would benefit from the ability to push a large amount of flow from s to b in a single step. This idea gives the intuition behind the *push/relabel* algorithm.

More precisely, in the push-relabel algorithm, we will be able to push K units of flow at one time from s to b. If the total capacity from b to t is then less than K, we will push what we can across those lengths, and then send whatever excess remains at b back to s.

For this purpose, we introduce the concept of *preflow*. Preflow has to satisfy capacity constraints. Instead of conservation constraints, we require that for each node v that is neither source nor sink, the amount of flow entering v is at least the amount of flow leaving v. The push-relabel algorithm works with preflow, slowly fixing the conservation constraints, while maintaining that there is no augmentation path from s to t. When the conservation constraints are finally satisfied, preflow becomes the solution to the max flow problem.

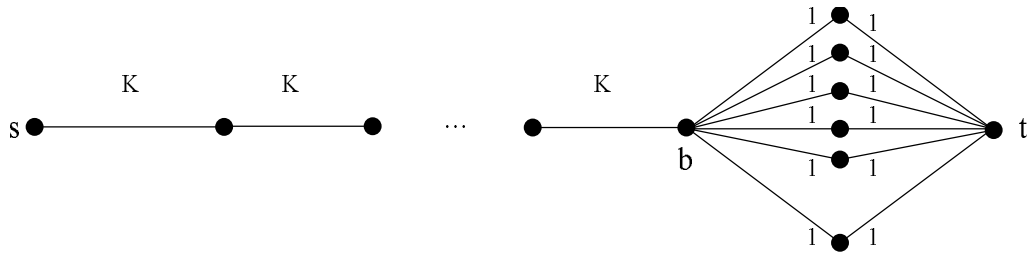


Figure 33: A bad case for algorithms based on augmentation

8.5.2 Description of Framework

The algorithm assigns to each node a value $d(v)$, called the *label*. We fix $d(s)$ at n and $d(t)$ at 0; these values never change. Labels of intermediate nodes are initialized to 0 and updated as described below during the algorithm. At any point in the algorithm, labels must satisfy the *label constraint*:

$$d(u) \leq d(v) + 1$$

for any residual edge (u, v) in the graph. Intuitively one can think of labels as water pressure. According to the laws of physics, water flows from high pressure to the low pressure. This algorithm follows an analogous principle, it pushes flow from a higher label to a lower label.

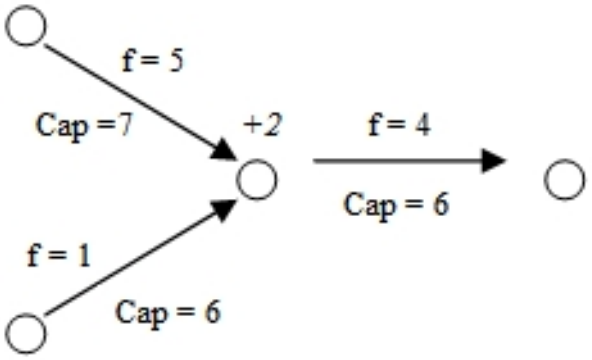


Figure 34: A portion of a graph with excess

The algorithm begins by saturating each of the edges at the source with preflow. This creates excess at the nodes at the other end of these edges, meaning there is more flow entering than leaving them. This also saturates all edges going out of the source, therefore there are no residual edges from source to any node in the residual graph. Hence the label constraint is satisfied. The above-mentioned violation of the conservation constraint allowed by preflow is precisely this excess; violation in the other direction—that is, more flow leaving than entering a node other than the source—will not occur. Nodes with excess are called *active nodes*.

Push-relabel algorithm tries to "fix" active nodes, by trying to push excesses to sink. A push operation can be thought of moving excesses in the graph. For example, consider the portion of a graph in Figure 34, after a one unit push from the center node to the right, the residual graph looks like Figure 35 after a two unit push in the same direction the residual graph is like Figure 35. Note that we cannot push any further because of capacity constraints. The first is an example of a *nonsaturating push* and the second is an example of a *saturating push* since the residual edge is saturated.

From this point, we proceed to push preflow across residual edges in the graph until no active nodes remain, at which point we have reached a final state. In each of these pushes, we move as much flow as possible without exceeding either the capacity of the residual edge or the quantity of excess on the node we are pushing from. Further, we may only push from a node v to another node u if the two nodes satisfy the equation:

$$d(v) = d(u) + 1$$

This restriction maintains the label constraint for the resulting (u, v) residual edge. An edge that satisfies the restriction is called *admissible*.

It is evident that, after the initial pushes from the source described above, no further pushes are immediately possible, since all interior nodes were initially labeled with a value of 0—clearly, we would be unable to push from any node v with excess, since $d(v)$ is 0 and there is no node u with $d(u) = -1$.

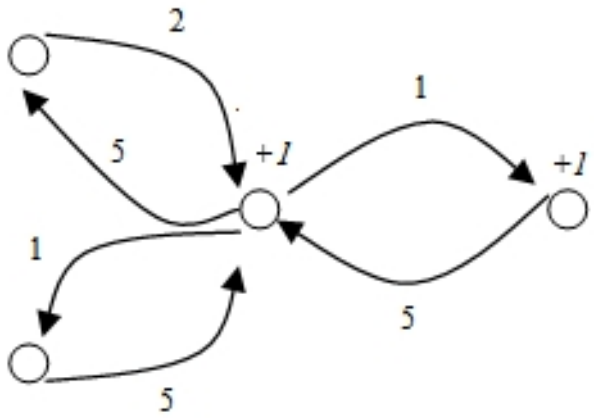


Figure 35: Residual graph after a nonsaturating push

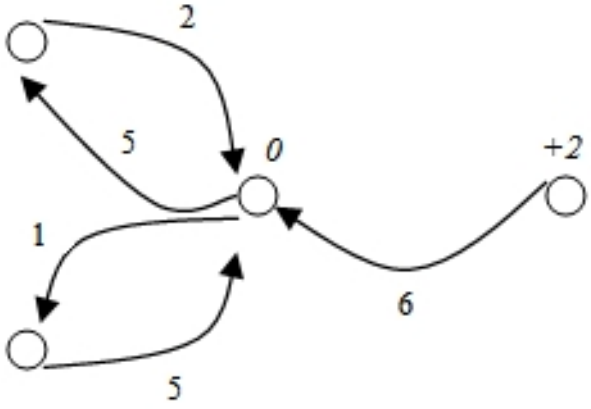


Figure 36: Residual graph after a saturating push

The next step, therefore, is to relabel nodes. When relabeling, we increase the labels of all active nodes as far as possible without violating the label constraint, i.e. we set the label of v to the minimum label of the neighboring nodes plus 1. Formally,

$$d(v) \leftarrow 1 + min(d(u))$$

where (v,u) is a residual edge

Relabeling a single active node v increases its label to precisely the point where it is possible to push something over a residual edge, since after relabeling, we must have a residual edge to some w such that $d(v) = d(w)+1$. Otherwise, either $d(v)$ would be less than $d(w)+1$ for all residual (v, w) edges, in which case we could further increase the label of v, or $d(v)$ would be greater than $d(w) + 1$ for some residual (v, w) edge, and we would be violating the label constraint. Thus, the rules we have given for relabeling are exactly what is needed to allow further pushing.

It is worth observing that at any time after the first set of pushes, a node's label provides a lower bound on the shortest-path distance from that node to the sink in the residual graph. If $d(v) = n$ for some v, there are at least n residual edges between v and t, since $d(t) = 0$ and across each residual edge we have a drop of no more than 1 in the value of the label. We will make use of this fact later in our discussion.

What is described above is the entirety of the algorithm; we can summarize its execution as follows:

1. Set the source label $d(s) = n$, the sink label to $d(t) = 0$, and the labels on the remaining nodes to $d(v) = 0$.
2. Send out as much flow as possible from the source s, saturating its outgoing edges and placing excesses on its neighboring nodes.
3. Calculate the residual edges.
4. Relabel the active nodes, increasing values as much as possible without violating the label constraint (i.e. set the label to the minimum label of the neighboring nodes plus 1)
5. Push as much flow as possible on some admissible edge.
6. Repeat steps 4 and 5 until there are no active nodes left in the graph.

We will show in a later section that the above algorithm eventually terminates with no excesses on any nodes except the source and the sink, and that it has calculated a maximum flow.

8.5.3 An Illustrated Example

In figure 37 we show a graph where initially the labels are set to: $d(s) = n$, $d(t) = 0$ and $d(v) = 0$.

We first saturate all (s, u) edges (ie. all edges originating from the source), update the excesses on the nodes and create the residual edges (figure 8.5.3).

Next, we update the labels on the nodes in such a way that will allow us in the next step to push flow from an active node. Thus the question now becomes 'Which arcs can I push on?' To answer this question consider the following cases (all the arcs (v, w) have $d(v) \leq d(w) + 1$):

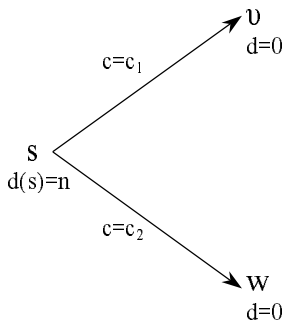
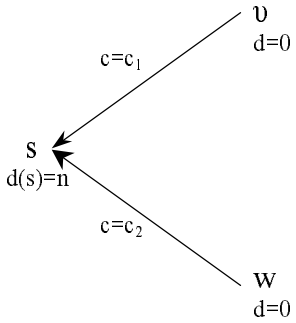
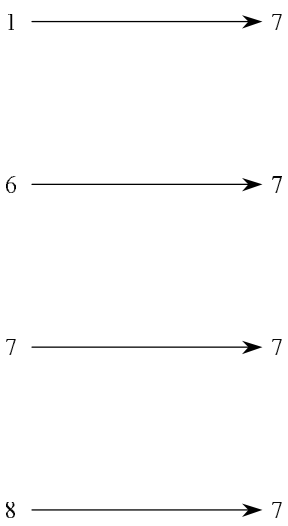


Figure 37:





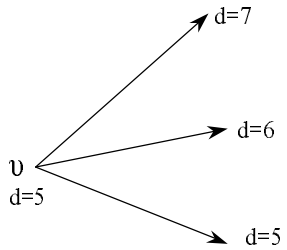


Figure 8.5.3: I cannot push the arc from 1 to 7, since it would introduce a residual edge from 7 to 1, violating the labeling constraint.

Figure 8.5.3: For the arc 6 to 7, we can push a plus. However, we will not allow pushes in this case, as we do not want to allow what are essentially 'backwards' pushes — we are trying to push from higher to lower labels since we want to push traffic closer to the sink node.

Figure 8.5.3: For the arc 7 to 7, we could push here without violating the labeling constraints, but allowing this would let the algorithm push flow back and forth on the edge repeatedly, constituting a cycle with no progress being made.

Figure 8.5.3: For the arc 8 to 7, we can legally push here, and in fact, this is the only case we will allow pushes from $d + 1$ to d. Now, we can see why we disallow pushes on the arc of the form $6- > 7$. These two types of pushes together, would allow flow to be pushed back and forth repeatedly with no progress.

From the above it is clear that we can't push any flow from v in figure 8.5.3 before relabeling it. After relabeling v gets set to 6 here, since it cannot go higher without violating labeling constraints.

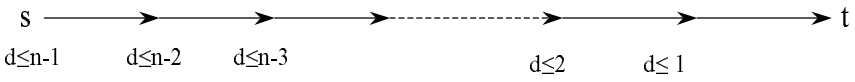
8.5.4 Analysis of the Algorithm

Liveliness We first prove that the algorithm can always perform a push or relabel step as long as there is an active node; i.e., as long as there is some node other than the source and sink with an excess. This is a liveliness property for the algorithm. Note first that an active node v must have at least an outgoing edge in the residual graph; such an edge was introduced when the excess came in. We then see that we must be able either to push from an active node or to relabel it:

- If there exist a node connected to v by an outgoing edge that has label $d(v) - 1$ then we can push.
- If the previous statement is not true, all nodes connected to v must have a label $> d(v) - 1$. In this case we can increase v's label.

Since some operation can always be performed on an active node, it is clear that the algorithm is live at all points in execution–that is, we cannot possibly reach a point prior to completion where no further action is possible. If we reach a state where we can neither push or relabel anything, there are no active nodes, so we have reached the end of the algorithm. At this time, all excesses must have been pushed either to the sink or back to the source.

Correctness of the Algorithm Why does this algorithm produce the maximum flow? To answer this question, we first prove that throughout the execution of the push-relabel algorithm, the manner in



which labels are maintained ensures that there is no augmenting flow path from the source to the sink.

Consider any path from s to t. Since any such path can be at most of length $n-1$, and we have the labeling constraint, we arrive at the conclusion that $d(s) <= n-1$. But we know that $d(s) = n$ and $d(t) = 0$. This is a contradiction, so we have no augmenting paths (figure 8.5.4).

So, throughout the execution of the algorithm, there is no augmenting path from the source to the sink. Since our algorithm only stops when all excesses are at the source or the sink, when the algorithm stops, there are no excesses in the graph and no augmenting path, i.e. we have a max flow. So to show correctness, we need to prove that the algorithm does indeed stop. We will do this by bounding the number of relabel operations and the number of pushes. To do this, we will need to draw a distinction between saturating pushes—those which fill the capacity of a residual edge—and non-saturating pushes, which do not. The latter occur in the case where there is more capacity on the edge over which we are pushing than there is excess on the node we are pushing from.

We begin by bounding the number of relabel operations.

Bounding Label Sizes We will show that the labels of nodes are bounded in the push-relabel algorithm. We begin by proving the following theorem which guarantees the existence of a residual path from an active node to s. This will be used in bounding label sizes later.

Theorem 8.11. *For any active node v, there is a simple residual path [4] from v to the source s.*

Proof: As a first attempt to prove this, one could try using induction. Assume there is a residual path from some active node w to the source and that there is a residual edge from w to v. Then when we push from w to v, v becomes active and a residual edge is created in the opposite direction; this edge attaches to the beginning of the residual path from w to the source, and so we have a path from v to the source. Unfortunately, this proof will get rather complicated since there are many special cases (e.g. what if later in the algorithm, one of the residual edges on that path from v to the source is saturated by a push?) to be considered. A more elegant proof using contradiction is the following.

Assume that there does not exist such a path from v to the source s. Define A to be the set of nodes reachable from v in E_f (the graph of non-0 residual edges). $s \notin A$. $\bar{A}$ is the rest of the nodes. By definition, since there are no deficits in the graph,

$$\forall(w \neq S), E_f(w) \geq 0)$$

where $E_f(w)$ is the excess on node w, i.e. the flow into w minus the flow out of w. This implies that

$$E_f(A) > 0, \tag{13}$$

since $v \in A$ and $E_f(v) > 0$, since it has an excess.

Consider some node x in A and some node w in $\bar{A}$. There are two possible edges (in the original graph) that go between x and w:

[4] A simple path is one in which no node is used twice; we can easily convert any non-simple path to a simple one by eliminating useless cycles.

- The edge (w, x). We will prove that the flow on this edge is 0. Suppose there is positive flow on this edge. Then there would be a residual edge from x to w. Since $x \in A$, x is reachable from v which implies that w would be reachable from v. But then w would be in A by the definition of A. This gives a contradiction. Hence, the flow from w to x on this edge must be 0: $f[(w, x)] = 0$.
- The edge (x, w). This edge must be saturated with flow, otherwise w would be reachable from x, which is impossible by the argument for the previous case. The flow from w to x on this edge is the negative of this saturation amount: $f[(w, x)] < 0$.

Therefore,

$$\forall w \in \bar{A}, x \in A : f[(w, x)] \leq 0. \tag{14}$$

By the definition of E_f,

$$E_f(A) = \sum_{x \in A} E_f(x) = \sum_{x \in A} \sum_{(w,x)} f[(w, x)].$$

When w and x are both in A, the contribution of the edge between them to the summation will cancel, since in one term it will be positive, and in the other it will be negative. Therefore, we only need to consider the contribution of edges (w, x) where $x \in A$ and $w \in \bar{A}$. Considering only those flows and substituting inequality 14 gives:

$$E_f(A) = \sum_{x \in A, w \in \bar{A}} f[(w, x)] \leq 0.$$

This contradicts inequality 13. Therefore, our assumption was invalid and there must be a residual path from v to S.

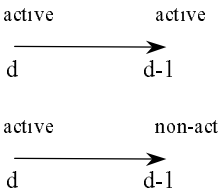
Aside: Does A include the sink? It may include the sink, but it does not matter for this proof, since $\sum E_f \geq 0$ even in this case. Excesses are not really "cancelled" at the sink; instead, they are stockpiled there and a large stockpile at the sink is good. We will now use this theorem to prove that label sizes are bounded.

Theorem 8.12. *The label of a node is at most 2n.*

Proof: Consider an active node v with label d. By theorem 8.11, there is a simple path in the residual graph from v to S. Along a residual path, the label between successive nodes can decrease by at most 1, from the label constraint. If the path from v to S has length k, then $d_S \geq d - k$. Since $d_s = n$ and $k \leq n$, $n \geq d - n$. This implies that $d \leq 2n$. The same limit holds for inactive nodes since they were active when they got relabelled.

This label limit gives a bound on the number of relabel operations that can occur. There are $O(n)$ nodes, each with at most $2n$ relabels, giving a bound of $O(n^2)$. ∎

Bounding the number of saturating pushes Consider an edge in the graph. Imagine a series of saturating pushes back and forth across this edge. The first saturating push sends from label d to label $d - 1$. After this, the second push requires a relabel of $d - 1$ to $d + 1$ to enable the reverse push. Each push after the first requires a relabeling of one of the nodes. Theorem 8.12 implies that the label can reach at most $2n$, so there at most $O(n)$ saturating pushes per edge. This implies that there are at most $O(nm)$ saturating pushes for the algorithm.



Bounding the number of non-saturating pushes Let us first convince ourselves that the above analysis for saturating pushes will not work for bounding the number of non-saturating pushes. The reason is that we can keep sending little non-saturating pushes across an edge without raising the label (if more excess arrives at the from-node). But, it seems that the algorithm is making progress, since intuitively, it is pushing the excess towards the sink (lower label).

In order to prove this formally, we will need the notion of a *potential function*. Potential functions are a useful technique for analyzing algorithms in which progress is being made towards the optimal solution. We will define a potential function Φ which describes how far the current state is from the optimal. Then we will analyze the contribution of each operation in the algorithm to the potential function. There are many potential functions one could choose for this algorithm; we will choose a rather simple one which nevertheless allows us to prove a bound on the number of non-saturating pushes.

Define the potential function to be the sum of all labels of active nodes:

$$\Phi = \sum_{v \ active} d(v)$$

This potential function cannot be negative since all labels are non-negative. We will show that relabels and saturating pushes make a finite total positive contribution to the potential function. Non-saturating pushes will have a negative contribution to Φ associated with each push. Therefore, after some number of non-saturating pushes, Φ will become negative; since it cannot be negative, the algorithm must stop before performing that many non-saturating pushes. Let us analyze $\Delta\Phi$, the change in Φ due to the various operations that the algorithm can perform.

- **Relabels:** For a single relabel step, $\Delta\Phi$ = amount of relabeling. The total amount of relabeling per node is $2n$. Therefore, the maximum contribution to Φ from relabeling all nodes is $\leq n * 2n$, which is
$$\Delta\Phi(all\ relabels) \leq O(n^2).$$
- **Saturating pushes:** A saturating push from v to w may potentially add w to the list of active nodes, if it was previously inactive (if v gets inactive after the push, we will treat it as a non-saturating push for the purposes of our analysis). Therefore, a saturating push may increase the potential function by as much as $2n$ (the max value for a label). There are at most $O(nm)$ saturating pushes, from 8.5.4. Therefore, the total change due to saturating pushes is
$$\Delta\Phi(all\ sat.\ pushes) \leq O(n^2 m).$$
- **Non-saturating pushes:** There are two kinds of non-saturating pushes: active-to-active and active-to-inactive. See figures 8.5.4 and 8.5.4.
  - In the first case, the node with label d becomes inactive (all of its excess was pushed); the other node changes neither activity nor label. Thus $\Delta\Phi = -d$

– In the second case, the node with label d becomes inactive and the node with label $d-1$ changes from inactive to active. Thus $\Delta\Phi = -d + (d-1) = -1$

Therefore, for each non-saturating push, $\Delta\Phi(one\ non-sat.\ push) \leq -1$.

We have found that saturating pushes and relabels contribute $O(n^2m)$ total to the potential function. Each non-saturating push lowers the potential function by at least 1. Therefore, since potential function does not become negative, there can be no more than $O(n^2m)$ total non-saturating pushes.

Running time analysis Since the number of pushes and relabels that the algorithm performs is bounded, the algorithm does indeed stop. From the discussion in 8.5.4, this proves that the algorithm is correct, i.e. it does produce a max flow. Our bounds on the number of operations also allow us to bound the running time of the algorithm. The number of relabels and saturating pushes are dominated by the number of non-saturating pushes, $O(n^2m)$. So, we can consider the number of steps to be $O(n^2m)$. Each step takes $O(n)$ time, since we need to search the entire graph of n nodes for an active node, and then search its neighbors (at most n) for a place to which to push or to determine the value of the new label. The total running time is $O(n^3m)$. We can improve though the running time of the push/relabel algorithm using a better implementation.

8.5.5 A better implementation: Discharge/Relabel

We mentioned earlier that the running time obtained for the naive implementation of push-relabel can be improved by using different rules for ordering the push and relabel operations. We describe one such ordering rule; the resulting algorithm is called discharge/relabel. Recall that the push-relabel algorithm performed $O(n^2m)$ operations, and the time per operation was $O(n)$. The bottleneck in the analysis was the analysis of non-saturating pushes: there were at most $O(n^2m)$ of them, and each took $O(n)$ time.

In fact, we can reduce the time per operation as follows. First, we can easily maintain a list of active nodes, eliminating the search for an available active node. Second, we can relabel and do lots of pushes from a single active node; after those pushes, either there is an excess, in which case relabeling this node is possible, or there is no excess, in which case we just move on to the next active node in our list. Note that the push operation may create newly active nodes. All such nodes are added to the list of active nodes. The two actions performed by the algorithm are relabeling and discharging. A discharge takes a node and keeps pushing flow out of it until no more pushes are possible. At this point, either the node has no excess, or relabeling the node is possible. The discharge/relabel algorithm can be implemented so that the running time is $O(n^2m)$; an improvement of a factor of n over the running time of the naive algorithm. However we have to be a little careful in the analysis in order to claim this bound.

Running Time of Discharge/Relabel We will use the discharge/relabel algorithm as an example to demonstrate the method of analyzing the time complexity of a non-trivial algorithm. For simple iterative algorithms, we are accustomed to calculating complexity by breaking the algorithm down into its iterative steps, phases, and so on, then multiplying the number of steps by the complexity of each. However, algorithms such as discharge/relabel do not have as clear of an iterative structure, so a similar analysis may be impossible or uninformative. Instead, we will break algorithms like this down into the different kinds of "work" that they do, and we will analyze the total running time of each type of work.

The actions of the discharge/relabel algorithm can be broken down into the following types of work:

- Relabel

- Saturating push
- Nonsaturating push
- Choosing the next admissible edge
- Finding the next active node

For each type of work, we will find the per-operation complexity, then find the total complexity for that type of work.

Relabel Consider a relabel operation. When we relabel a node, we must examine all of the edges connected to it to find the new label. Therefore, the relabel step uses time proportional to the degree of the node. The total work is therefore the sum over all nodes of the product of the degree of the node and the number of times it can be relabelled, which is $O(n)$. The degree summed over all nodes is proportional to the total number of edges, or $O(m)$. The total work is thus $O(mn)$:

$$\sum_v \text{degree}(v)O(n) = O(mn) \tag{15}$$

Note, however, that this expression takes into account only operations that actually result in a change of some node label, not work done to see if a node label can or must be changed. This will be discussed further below.

Saturating Push Next, consider a saturating push. Such a push can be done in constant time, given that we know which nodes and edge will be involved in the push. This is valid because we've pulled out the work of finding the next admissible edge (to be analyzed below). The total number of saturating pushes will be $O(mn)$ because each of the m edges participates in at most $O(n)$ saturating pushes. The total complexity of saturating pushes is thus $O(mn)$.

Non-saturating Push Now, we can analyze the complexity of non-saturating pushes. Given that we know which nodes and edge participate in the push, we only need constant time. It was shown in 8.5.4 that the number of non-saturating pushes is $O(n^2m)$, so this is the total complexity of non-saturating pushes.

Next Admissible Edge and Next Active Node The issue which has been sidestepped until now is this: how do we keep track of useful edges so that we can actually do the pushes in $O(1)$ time and so that we do not waste work checking to see if nodes need to be relabelled?

The answer is to maintain a linked list of admissible edges (ones we are able to push on) for each node. When we need to do a push, we simply take the next edge on the list, remove it from the list, and check whether it is still admissible (the other end of this edge might have been relabelled since we put this edge into the list). If the edge is still admissible, we push on it. If it is not admissible, we go to the next edge on the list. This takes $O(1)$ time per edge.

What about the time to build the list? It can be lumped into the time for a relabel operation. Observe that if we exhaust the list of useful edges, we can be assured that a relabel operation will be successful (*i.e.*, that it will result in a change of the node label). Thus, when we exhaust a list, we relabel

and build a new list at the same time, since both involve examining the edges from the current node. This also ensures that we do not waste work examining edges to see if a node needs to be relabelled without actually relabeling it.

Going back to our complexity analysis of different work types, we can see that finding the next admissible edge takes $O(1)$ time if we just move down the linked list. Finding the next active node can be done in $O(1)$ time if we maintain a linked list of active nodes. Since we execute either a relabel or a push when we find an active node, the work of finding the node can be lumped together with these operations.

Total Complexity Table 1 summarizes the per-operation and total complexity of each type of work. Finally, we see that the complexity of the whole algorithm is $O(n^2m)$. This example is instructive because non-trivial algorithms will generally require the use of this method for determining time complexity. We have also seen that going through this analysis suggests how to set up data structures in order to obtain the claimed running time for the algorithm.

Table 1:

`Type of Work`	`Per Operation`	`Total`
`Relabel`	$O(degree)$	$O(mn)$
`Saturating Push`	$O(1)$	$O(mn)$
`Non-saturating Push`	$O(1)$	$O(n^2m)$
`Next Admissible Edge`	$O(1)$	
`Next Active Node`	$O(1)$	

Can We Do Better? With data structures called dynamic trees (which are beyond the scope of this class), the complexity can be reduced to $O(nm \log(n^2/m))$ (see the original Goldberg-Tarjan paper [5]). Also, in the special case that the graph has unit capacities everywhere, all pushes are saturating pushes and discharge/relabel runs in $O(mn)$ time. Note that Ford-Fulkerson also runs in $O(mn)$ time on such a graph because the flow can be at most m and there can be at most n augmenting paths.

8.6 Flow with Lower Bounds

8.6.1 Motivating example - "Job Assignment" problem

The fact that given integer capacities one can always find integral max-flow can be used to apply max-flow formulation to a variety of optimization problems. For example, consider the problem of assigning jobs to people. Each person can perform some subset of the jobs and cannot be assigned more than a certain maximum number of jobs. The objective is to find a valid assignment of jobs to people that maximizes the number of assigned jobs. Note that the statement of the problem clearly implies that we would like to get an integral solution.

We can pose this as a maximum flow problem. Suppose the jobs are $\{J_1, J_2, \cdots, J_m\}$, and the available people are $\{P_1, P_2, \cdots, P_n\}$. Corresponding to each person P_j is the value x_j, the maximum number of jobs that the person can handle.

We construct a bipartite graph with a vertex corresponding to each job J_i on the left side, and a vertex corresponding to each person P_i on the right side. If the job J_i can be assigned to person P_j, we add an edge J_iP_j with capacity 1.

Now we add two more vertices to our bipartite graph. First, we add a source s and for each $i \in \{1 \cdots m\}$ connect it to J_i with edge capacity 1. Then we add a sink t and for each $j \in \{1 \cdots m\}$ connect P_j to t with edge capacity x_j. (See Figure 38).

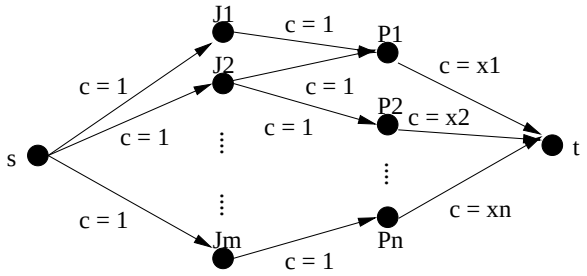


Figure 38: Transforming job assignment into flow.

Observe that an integral flow in this graph corresponds to a valid assignment of jobs to people. Each edge J_iP_j with flow of 1 corresponds to the assignment of job J_i to person P_j. The value of the flow corresponds to the number of assigned jobs. Since all edge capacities are integral, the maximum flow is integral as well.

Note that if each person can handle at most 1 job, then this problem is equivalent to the maximum matching problem in the subgraph induced by $V - \{s, t\}$ (the original bipartite graph of jobs and people).

Let us slightly generalize the problem. Suppose that in addition to the specification of the simple job assignment problem, we add the following conditions:

1. certain jobs must be assigned;
2. certain people must be assigned some minimum number of jobs.

These conditions lead to a new constraint in the corresponding graph formulation; now we also have a **lower bound constraint** on certain edges, i.e. the flow on those edges must be at least a certain minimum amount.

Figure 39: Lower Bounds To Excesses And Deficits.

For each job J_i that must be assigned, we add the constraint that the flow on sJ_i be at least 1. For each person P_j that must be assigned a minimum number of jobs we add the constraint that the flow on edge $P_j t$ must be at least the minimum number of jobs that person must be assigned.

8.6.2 Flows With Lower Bound Constraints

The job assignment example leads us to consider the more general problem of flows with both upper and lower bounds on edges.

Suppose we are given a graph G with capacity $c(ij)$ and lower bound flow $l(ij)$ for each edge ij. Now the flow $f(ij)$ through the edge ij must satisfy $l(ij) \leq f(ij) \leq c(ij)$ for all edges ij.

We will show that this problem can be reduced to max flow.

Up until now we worked with flows that satisfied conservation constraints. In order to solve flow problems with lower bounds on the capacities we will have to work with flows that violate conservation constraints (while satisfying capacity constraints).

Given a max flow problem with lower bound constraints, we solve it in three steps. We will discuss each one of these steps in detail in the subsequent sections.

1. Translate the lower bound constraints into excesses and deficits: for each edge ij, we set $f(ij) = l(ij)$ and update the capacity by setting $c(ij) = c(ij) - l(ij)$. Do not add the corresponding residual edge.

2. Add auxiliary s' and t' nodes. For every v with excess add an edge $s'v$ with capacity equal to the excess. For each v with a deficit add vt' with capacity equal to that deficit.

   Find the max flow from s' tot' in the resulting graph.

3. Compute max flow between the original s and t in the residual graph obtained at step 2 after deleting s', t', and all edges adjacent to them.

Translating Lower Bound Constraints Into Excesses And Deficits: In the first step of the algorithm we set the flow on each edge uv to be equal to the lower bound on this edge, $l(uv)$. This adds a (positive) excess to v and (negative) deficit to u. We also update the capacity of uv, reducing it by $l(uv)$. Notice that we do not add a backwards residual edge because no flow can be pushed back without violating the lower bound. An example shows the transformation.

Removing Excesses And Deficits: Denote the flow vector that we got in Step 1 by f_1. The goal of the second step is to compute flow f_2 such that $f_1 + f_2$ satisfies both capacity (including lower bounds) and conservation constraints. Observe that, for any flow vector f that satisfies the residual capacities computed in Step 1, $f_1 + f$ will satisfy the capacity constraints of the original problem. The challenge is to find f_2 such that $f_1 + f_2$ will satisfy the conservation constraints as well.

We add an auxiliary source s' and sink t'. Connect s' to all nodes with excess by edges with capacity equal to the excess, and connect t' to all nodes with deficit with capacity equal to the deficit. We also connect s to t and t to s with infinite capacity edges. (The importance of these edges will be seen later). Now we compute max flow from s' to t' in the resulting graph.

Let $\tilde{f}$ be the result of the above max flow calculation. We claim that:

- $\tilde{f}$ saturates all the edges outgoing from s' and all the edges incoming to t' if and only if there is a solution to the original problem.
- In this case, f_2 can be obtained by considering the coordinates of $\tilde{f}$ that correspond to the original graph edges.

First, assume that we found $\tilde{f}$ that saturates all the edges adjacent to s' and t'. Consider f_2 that is obtained by restricting $\tilde{f}$ to the edges in the original graph. If node v has excess $Ex(v)$ then there is an edge $s'v$ with $\tilde{f}(s'v) = Ex(v)$. Since v has an excess there is no vt' edge. Consider the conservation constraint at v in f_2. From the point of view of v, the only difference between $\tilde{f}$ and f_2 is that the edge $s'v$ (together with its flow) disappeared. Hence, the incoming minus the outgoing is equal to $-Ex(v)$. But the same difference for f_1 was $Ex(v)$, which means that $f_1 + f_2$ will satisfy conservation constraints. A similar argument works for nodes with deficit in f_1.

Now we need to show that existence of a feasible solution to the original problem implies that $\tilde{f}$ will saturate all the edges adjacent to s' and t'. Let f^* be a feasible solution to the original problem. Observe that $f^* - l$ is feasible with respect to residual capacity constraints computed in Step 1. Consider the difference between incoming and outgoing flow in $f^* - l$ for some node v:

$$\begin{aligned}\sum_{uv\in E}(f^*(uv) - l(uv)) - \sum_{vu\in E}(f^*(vu) - l(vu)) &= \sum_{uv\in E} f^*(uv) - \sum_{vu\in E} f^*(vu) - \sum_{uv\in E} l(uv) + \sum_{vu\in E} l(vu) \\ &= -\sum_{uv\in E} l(uv) + \sum_{vu\in E} l(vu)\end{aligned}$$

But the last expression is exactly equal to $-Ex_{f_1}(v) = -c(s'v)$. In other words, if v has an excess, then $f^* - l$ has a deficit of exactly the same value at v. Now extend $f^* - l$ by saturating all the edges adjacent to s' and t'. This process cancels all the excesses and deficits giving a legal flow in G'. This flow is maximum, since the cut around s' (and around t') is saturated. Hence, we proved that there exists a flow that saturates these cuts, which means that the flow $\tilde{f}$ that we will find will saturate these cuts as well.

Why did we add the infinite capacity edges connecting s' and t' ? Figure 40 shows an example where there is no $\tilde{f}$ that saturates all the edges adjacent to t' and s'. At the same time, it is easy to see that a feasible flow that satisfies the lower bound exists. The reason is that we did not add the infinite capacity edges in this example. Adding these edges solves the problem, as we can see from Figure 41. It is an interesting exercise to go over the above proof and try to find the place where we have used existence of these edges.

Max Flow On Residual Graph: The result of the first two steps is a flow f_1+f_2 that satisfies capacity and conservation constraints. Let f^* be some $s-t$ max flow that satisfies capacity and conservation constraints. It is easy to see that $f^* - f_1 - f_2$ is a feasible flow in the residual graph with capacities equal to $c(uv) - f_1(uv) - f_2(uv)$ for $uv \in E$ and $f_1(uv) + f_2(uv) - l(uv)$ for $vu \in E$. Thus, as the last step of the algorithm, we compute these residual capacities and compute flow f_3, which is max $s-t$ flow in this residual graph. The answer to the original problem is $f_1 + f_2 + f_3$.

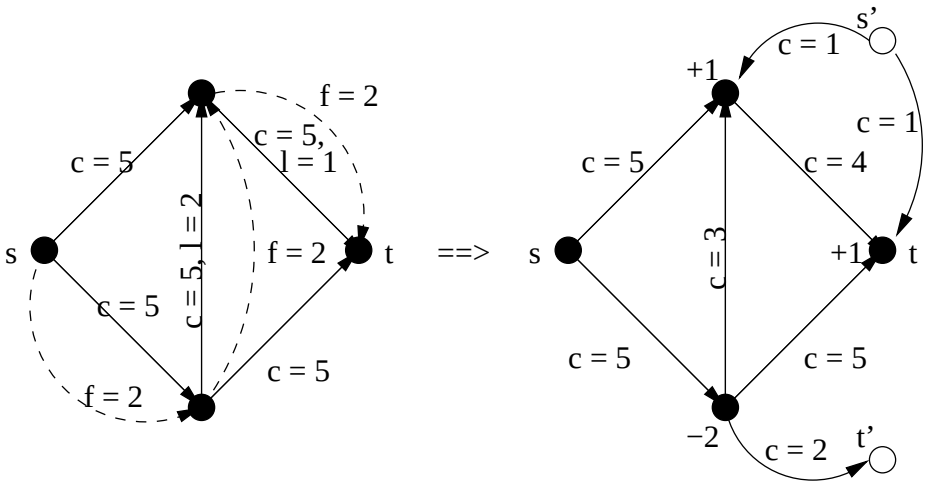


Figure 40: The original graph with two edges that have lower bound constraints has a solution (a legal flow) as shown by dashed lines. However the transformed graph cannot remove all excesses and deficits by finding a flow from s' to t'.

Effects of Lower Bound Constraints Introducing lower bounds can change the properties of the max flow algorithm in somewhat unexpected ways. Previously, there always existed a feasible zero flow solution (all edges carry no flow). With lower bound constraints, the zero flow may not be feasible. In fact, in certain instances, the flow may have a negative maximum value (that is, the net flow is from sink to source rather than vice versa), a situation which would not occur with the original max flow algorithm, under the assumption that no capacity can be negative.

Consider the following example (see Figure 43). We have a graph where the forward capacities from s to t are small relative to the backward capacities from t to s, and one of the backward edges has a large lower bound constraint. When we remove excesses and deficits, we introduce a large flow from t back to s. Thus, when we begin calculating the maximum flow with the residual graph, we are starting with a large negative (but feasible) flow that no forward flow could possibly cancel. The value of the maximum $s - t$ flow in this example is -23, so there is a net flow backwards from t to s.

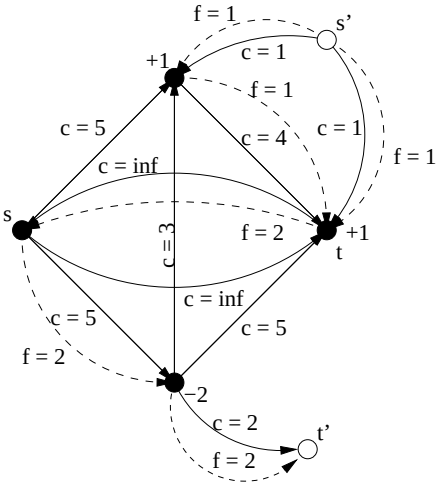


Figure 41: Adding edges from original source to sink and back allows us to find a legal flow which removes the excesses and deficits. The solution is shown by dashed lines.

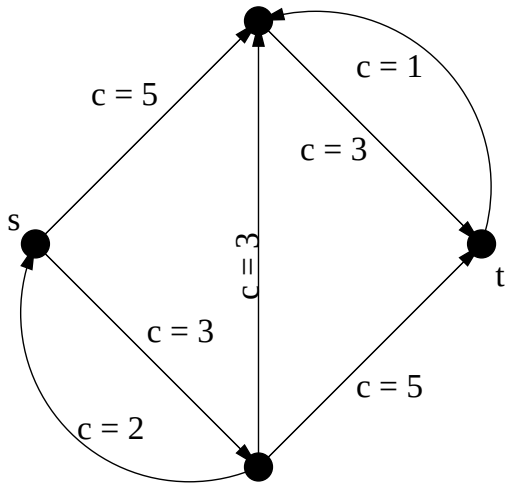


Figure 42: In the residual graph, we do not add residual flows against the lower bound flow, as pushing flow back over such edges would violate the lower bound constraints in the original graph.

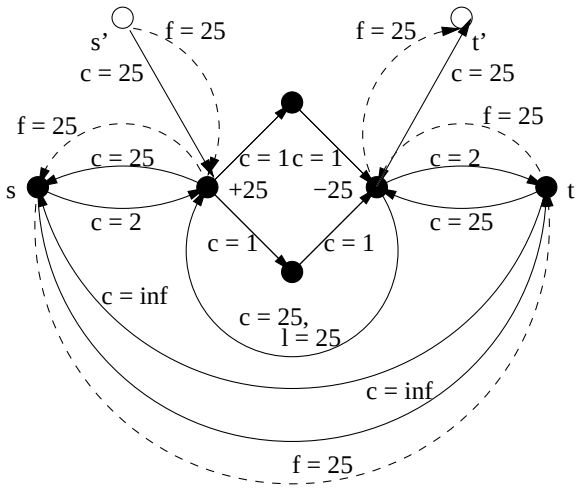


Figure 43: After removing the excesses and deficits from this graph, we are left with a large negative flow from the sink to the source, an impossible outcome with the original max flow algorithm. The flow is shown with dashed lines.