Chapter 2

Maximum Flow

We start by considering the following problem: suppose that we have built a network of pipes to transport oil in an area. Each pipe has a fixed capacity for pumping oil, measured by barrels per hour. Now suppose that we want to transport a very large quantity of oil from city s to city t. How do we use the network system of pipes so that the oil can be transported in the shortest time?

This problem can be modeled by the MAXIMUM FLOW problem, which will be the main topic of this chapter. The network of pipes will be modeled by a directed graph G with two distinguished vertices s and t. Each edge in the graph G is associated with an integer, indicating the capacity of the corresponding pipe. Now the problem is to assign each edge with a flow, less than or equal to its capacity, so that the maximum amount of flow goes from vertex s to vertex t.

The MAXIMUM FLOW problem arises in many settings in operations research and other fields. In particular, it can be used to model liquids flowing through pipes, parts through assembly lines, current through electrical networks, information through communication networks, and so forth. Efficient algorithms for the problem have received a great deal of attention in the last three decades.

In this chapter, we introduce two important techniques in solving the MAXIMUM FLOW problem. The first one is called the *shortest path saturation* method. Two algorithms, Dinic's algorithm and Karzanov's algorithm, are presented to illustrate this technique. The second method is a more recently developed technique, called the *preflow* method. An algorithm based on the preflow method is presented in Section 2.3. Remarks on related topics and further readings on the MAXIMUM FLOW problem are also given.

2.1 Preliminary

We start with the formal definitions.

Definition 2.1.1 A *flow network* $G = (V, E)$ is a directed graph with two distinguished vertices s (the source) and t (the sink). Each edge $[u, v]$ in G is associated with a positive integer $cap(u, v)$, called the *capacity* of the edge. If there is no edge from vertex u to vertex v, then we define $cap(u, v) = 0$.

Remark. There is no special restriction on the directed graph G that models a flow network. In particular, we allow edges in G to be directed into the source and out of the sink.

Intuitively, a flow in a flow network should satisfy the following three conditions: (1) the amount of flow along an edge should not exceed the capacity of the edge (capacity constraint); (2) a flow from a vertex u to a vertex v can be regarded as a "negative" flow of the same amount from vertex v to vertex u (skew symmetry); and (3) except for the source s and the sink t, the amount of flow getting into a vertex v should be equal to the amount of flow coming out of the vertex (flow conservation). These conditions are formally given in the following definition.

Definition 2.1.2 A *flow* f in a flow network $G = (V, E)$ is an integer-valued function on pairs of vertices of G satisfying the following conditions:

1. For all $u, v \in V$, $cap(u, v) \geq f(u, v)$ (*capacity constraint*);
2. For all $u, v \in V$, $f(u, v) = -f(v, u)$ (*skew symmetry*);
3. For all $u \neq s, t$, $\sum_{v \in V} f(u, v) = 0$ (*flow conservation*).

An edge $[u, v]$ is *saturated* by the flow f if $cap(u, v) = f(u, v)$. A path P in the flow network G is *saturated* by the flow f if at least one edge in P is saturated by f.

Note that even when there is no edge from a vertex u to a vertex v, the flow value $f(u, v)$ can still be non-zero. For example, suppose that there is no edge from u to v but there is an edge from v to u of capacity 10, and that the flow value $f(v, u)$ is equal to 8. Then by the skew symmetry property, the flow value $f(u, v)$ is equal to -8, which is not 0.

However, if there is neither edge from u to v and nor edge from v to u, then the flow value $f(u, v)$ must be 0. This is because from $cap(u, v) = cap(v, u) = 0$, by the capacity constraint property, we must have $f(u, v) \leq 0$

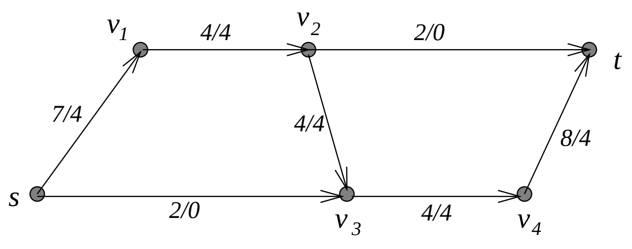

Figure 2.1: A flow network with a flow.

and $f(v,u) \le 0$. By the skew symmetry property, $f(v,u) \le 0$ implies $f(u,v) \ge 0$, which together with $f(u,v) \le 0$ gives $f(u,v) = 0$.

Figure 2.1 is an example of a flow network G with a flow, where on each edge $e = [u,v]$, we label a pair of numbers as "a/b" to indicate that the capacity of the edge e is a and the flow from vertex u to vertex v is b.

Given a flow network $G = (V,E)$ with the source s and the sink t, let f be a flow on G. The *value* of the flow is defined to be $\sum_{v \in V} f(s,v)$, denoted by $|f|$.

Now the Maximum Flow problem can be formally defined using our definition of optimization problems as a 4-tuple.

Maximum Flow $= \langle I_Q, S_Q, f_Q, opt_Q \rangle$

I_Q: the set of flow networks G

S_Q: $S_Q(G)$ is the set of flows f in G

f_Q: $f_Q(G,f)$ is equal to the flow value $|f|$

opt_Q: max

Our first observation on the properties of a flow is as follows.

Lemma 2.1.1 *Let $G = (V,E)$ be a flow network with the source s and the sink t, and let f be a flow in G. Then the value of the flow f is equal to $\sum_{v \in V} f(v,t)$.*

Proof. We have

$$|f| = \sum_{v \in V} f(s,v) = \sum_{w \in V} \sum_{v \in V} f(w,v) - \sum_{w \ne s} \sum_{v \in V} f(w,v)$$

By the skew symmetry property, $f(w,v) = -f(v,w)$. Note that in the sum $\sum_{w \in V} \sum_{v \in V} f(w,v)$, for each pair of vertices w and v, both $f(w,v)$ and

$f(v,w)$ appear exactly once. Thus, we have $\sum_{w\in V}\sum_{v\in V} f(w,v) = 0$. Now apply the skew symmetry property on the second term on the right hand side, we obtain

$$|f| = \sum_{w\neq s}\sum_{v\in V} f(v,w)$$

Thus, we have

$$|f| = \sum_{w\neq s}\sum_{v\in V} f(v,w) = \sum_{w\notin\{s,t\}}\sum_{v\in V} f(v,w) + \sum_{v\in V} f(v,t)$$

Finally, according to the skew symmetry property and the flow conservation property, for each $w \neq s,t$, we have

$$\sum_{v\in V} f(v,w) = -\sum_{v\in V} f(w,v) = 0$$

Thus, the sum $\sum_{w\notin\{s,t\}}\sum_{v\in V} f(v,w)$ is equal to 0. This gives the proof that $|f| = \sum_{v\in V} f(v,t)$. $\square$

The following lemma describes a basic technique to construct a positive flow in a flow network.

Lemma 2.1.2 *Let $G = (V,E)$ be a flow network with the source s and the sink t. There is a flow f in G with a positive value if and only if there is a path in G from s to t.*

PROOF. Suppose that there is a path P from the source s to the sink t. Let e be an edge on P with the minimum capacity $c > 0$ among all edges in P. Now it is easy to see that if we assign flow c to each edge on the path P, and assign flow 0 to all other edges, we get a valid flow of value $c > 0$ in the flow network G.

For the other direction, suppose that f is a flow of positive value in the flow network G. Suppose that the sink t is not reachable from the source s. Let V' be the set of vertices in G that are reachable from s. Then $t \notin V'$.

Let w be a vertex in V'. We first show

$$\sum_{v\in V'} f(w,v) = \sum_{v\in V} f(w,v) - \sum_{v\notin V'} f(w,v) \geq \sum_{v\in V} f(w,v) \qquad (2.1)$$

In fact, for any $v \notin V'$, since v is not reachable from the source s, there is no edge from w to v. Thus, $\mathit{cap}(w,v) = 0$, which implies $f(w,v) \leq 0$ by the capacity constraint property.

By Equation (2.1), we have

$$|f| = \sum_{v \in V} f(s,v) \leq \sum_{v \in V'} f(s,v) = \sum_{w \in V'} \sum_{v \in V'} f(w,v) - \sum_{w \in V' - \{s\}} \sum_{v \in V'} f(w,v)$$

By the skew symmetry property, $\sum_{w \in V'} \sum_{v \in V'} f(w,v) = 0$. Thus,

$$|f| = - \sum_{w \in V' - \{s\}} \sum_{v \in V'} f(w,v)$$

For each $w \in V' - \{s\}$, according to Equation (2.1) and the flow conservation property, we have (note $t \notin V'$ so w is neither t)

$$\sum_{v \in V'} f(w,v) \geq \sum_{v \in V} f(w,v) = 0$$

Thus, we have $|f| \leq 0$. This contradicts our assumption that f is a flow of positive value. This contradiction shows that the sink t must be reachable from the source s, or equivalently, there is a path in the flow network G from the source s to the sink t. $\square$

Thus, to construct a positive flow in a flow network G, we only need to find a path from the source to the sink. Many graph algorithms effectively find such a path.

Now one may suspect that finding a maximum flow is pretty straightforward: each time we find a path from the source to the sink, and add a new flow to saturate the path. After adding the new flow, if any edge becomes saturated, then it *seems* the edge has become useless so we delete it from the flow network. For those edges that are not saturated yet, it seems reasonable to have a new capacity for each of them to indicate the amount of room left along that edge to allow further flow through. Thus, the new capacity should be equal to the difference of the original capacity minus the amount of flow through that edge. Now on the resulting flow network, we find another path to add further flow, and so forth.

One might expect that if we repeat the above process until the flow network contains no path from the source s to the sink t, then the obtained flow must be a maximum flow. Unfortunately, this observation is incorrect.

Consider the flow network with the flow in Figure 2.1. After deleting all saturated edges, the sink t is no longer reachable from the source s (see Figure 2.2). However, it seems that we still can push a flow of value 2 along the "path" $s \to v_3 \to v_2 \to t$, where although we do not have an edge from v_3 to v_2, but we still can push a flow of value 2 from v_3 to v_2 by *reducing*

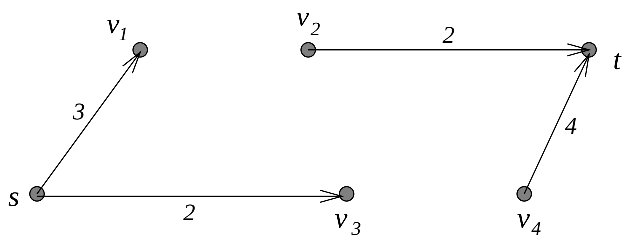

Figure 2.2: The sink t is not reachable from the source after deleting saturated edges.

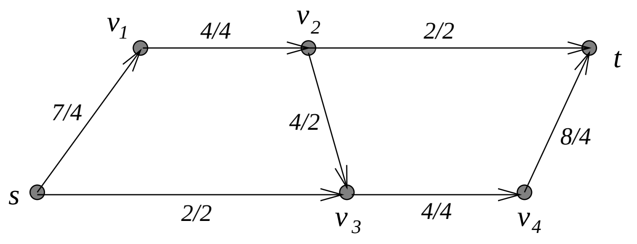

Figure 2.3: A flow larger than the one in Figure 2.1.

the original flow by 2 on edge $[v_2, v_3]$. This, in fact, does result in a larger flow in the original flow network, as shown in Figure 2.3.

Therefore, when a flow $f(u,v)$ is assigned on an edge $[u,v]$, it seems that not only do we need to modify the capacity of the edge $[u,v]$ to $cap(u,v) - f(u,v)$ to indicate the amount of further flow allowed through the edge, but also we need to record that a flow of amount $f(u,v)$ can be pushed along the opposite direction $[v,u]$, which is done by reducing the original flow along the edge $[u,v]$. In other words, we need add a new edge of capacity $f(u,v)$ from the vertex v to the vertex u. Motivated by this discussion, we have the following definition.

Definition 2.1.3 Given a flow network $G = (V,E)$ and given a flow f in G, the *residual network* $G_f = (V,E')$ of G (with respect to the flow f) is a flow network that has the same vertex set V as G. Moreover, for each vertex pair u,v, if $cap(u,v) > f(u,v)$, then $[u,v]$ is an edge in G_f with capacity $cap(u,v) - f(u,v)$.

Figure 2.4 is the residual network of the flow network in Figure 2.1 with respect to the flow given in the Figure. It can be clearly seen now that in the residual network, there is a path from s to t: $s \to v_3 \to v_2 \to t$.

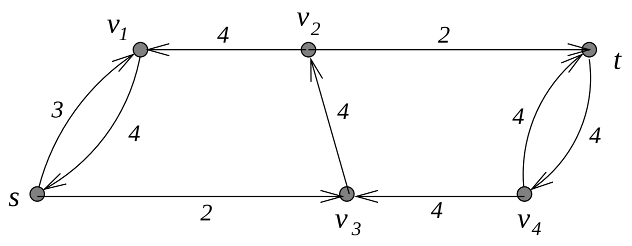

Figure 2.4: The residual network for Figure 2.1.

Remark. New edges may be created in the residual network G_f that were not present in the original flow network G. For example, there is no edge from vertex v_3 to vertex v_2 in the original flow network in Figure 2.1, but in the residual network in Figure 2.4, there is an edge from v_3 to v_2. However, if there is neither an edge from u to v nor an edge from v to u, then, since we must have $cap(u, v) = f(u, v) = 0$, there is also no edge from u to v in the residual network. This implies that the number of edges in a residual network cannot be more than twice of that in the original flow network. This fact will be useful when we analyze maximum flow algorithms.

Lemma 2.1.3 *Let G be a flow network and let f be a flow in G. If f^* is a flow in the residual network G_f, then the function $f^+ = f + f^*$, defined as $f^+(u, v) = f(u, v) + f^*(u, v)$ for all vertices u and v, is a flow with value $|f^+| = |f| + |f^*|$ in G.*

Proof. It suffices to verify that the function f^+ satisfies all the three constraints described in Definition 2.1.2. For each pair of vertices u and v in G, we denote by $cap(u, v)$ the capacity from u to v in the original flow network G, and by $cap_f(u, v)$ the capacity in the residual network G_f.

The Capacity Constraint Condition. We compute the value $cap(u, v) - f^+(u, v)$. By the definition we have

$$cap(u, v) - f^+(u, v) = cap(u, v) - f(u, v) - f^*(u, v)$$

Now by the definition of cap_f, we have $cap(u, v) - f(u, v) = cap_f(u, v)$. Moreover, since $f^*(u, v)$ is a flow in the residual network G_f, $cap_f(u, v) - f^*(u, v) \geq 0$. Consequently, we have $cap(u, v) - f^+(u, v) \geq 0$.

The Skew Symmetry Condition. Since both $f(u, v)$ and $f^*(u, v)$ are flows in the flow networks G and G_f, respectively, we have $f(u, v) = -f(v, u)$ and

$f^*(u,v) = -f^*(v,u)$. Thus,

$$f^+(u,v) = f(u,v) + f^*(u,v) = -f(v,u) - f^*(v,u) = -f^+(v,u)$$

The Flow Conservation Condition. Again, since both $f(u,v)$ and $f^*(u,v)$ are flows in the flow networks G and G_f, respectively, we have for all $u \neq s,t$

$$\sum_{v \in V} f^+(u,v) = \sum_{v \in V} f(u,v) + \sum_{v \in V} f^*(u,v) = 0$$

Thus, f^+ is a flow in the flow network G. For the flow value of f^+, we have

$$|f^+| = \sum_{v \in V} f^+(s,v) = \sum_{v \in V} f(s,v) + \sum_{v \in V} f^*(s,v) = |f| + |f^*|$$

□

Now we are ready for the following fundamental theorem for maximum flow algorithms.

Theorem 2.1.4 *Let G be a flow network and let f be a flow in G. The flow f is a maximum flow in G if and only if the residual network G_f has no positive flow, or equivalently, if and only if there is no path from the source s to the sink t in the residual network G_f.*

Proof. The equivalence of the second condition and the third condition is given by Lemma 2.1.2. Thus, we only need to prove that the first condition and the second condition are equivalent.

Suppose that f is a maximum flow in G. If the residual network G_f has a positive flow f^*, $|f^*| > 0$, then by Lemma 2.1.3, $f^+ = f + f^*$ is also a flow in G with flow value $|f| + |f^*|$. This contradicts the assumption that f is a maximum flow in G since $|f^*| > 0$. Thus, the residual network G_f has no positive flow.

For the other direction, we assume that f is not a maximum flow in G. Let $f_{\max}$ be a maximum flow in G. Thus, $|f_{\max}| - |f| > 0$. Now define a function f^- on each pair (u,v) of vertices in the residual network G_f as follows,

$$f^-(u,v) = f_{\max}(u,v) - f(u,v)$$

We claim that f^- is a valid flow in the residual network G_f.

The function f^- satisfies the capacity constraint condition: since $cap_f(u,v) = cap(u,v) - f(u,v)$, we have

$$cap_f(u,v) - f^-(u,v) = cap(u,v) - f(u,v) - f^-(u,v)$$

```
Algorithm.  Ford-Fulkerson

Input:  a flow network $G$
Output:  a maximum flow $f$ in $G$

1. let $f(u,v) = 0$ for all pairs $(u,v)$ of vertices in $G$;
2. construct the residual network $G_f$;
3. while there is a positive flow in $G_f$ do
       construct a positive flow $f^*$ in $G_f$;
       let $f = f + f^*$ be the new flow in $G$;
       construct the residual network $G_f$;
```

Figure 2.5: Ford-Fulkerson's method for maximum flow

Note that $f(u,v) + f^-(u,v) = f_{\max}(u,v)$. Since $f_{\max}$ is a flow in G, we have $cap(u,v) - f_{\max}(u,v) \geq 0$. Consequently, we have $cap_f(u,v) - f^-(u,v) \geq 0$.

The function f^- satisfies the skew symmetry condition:

$$f^-(u,v) = f_{\max}(u,v) - f(u,v) = -f_{\max}(v,u) + f(v,u) = -f^-(v,u)$$

The function f^- satisfies the flow conservation condition: for all $u \neq s, t$, we have

$$\sum_{v \in V} f^-(u,v) = \sum_{v \in V} f_{\max}(u,v) - \sum_{v \in V} f(u,v) = 0$$

Thus, f^- is a valid flow in the residual network G_f. Moreover, since we have

$$|f^-| = \sum_{v \in V} f^-(s,v) = \sum_{v \in V} f_{max}(s,v) - \sum_{v \in V} f(s,v) = |f_{max}| - |f| > 0$$

We conclude that the residual network G_f has a positive flow.

This completes the proof of the theorem. $\Box$

Theorem 2.1.4 suggests a classical method (called *Ford-Fulkerson's method*), described in Figure 2.5 for constructing a maximum flow in a given flow network.

According to Lemma 2.1.2, there is a positive flow in the residual network G_f if and only if there is a directed path from the source s to the sink t in G_f. Such a directed path can be found by a number of efficient graph search algorithms. Thus, the condition in the **while** loop in step 3 in the algorithm **Ford-Fulkerson** can be easily checked. Theorem 2.1.4 guarantees that when the algorithm halts, the obtained flow is a maximum flow.

The only problem left is how we construct a positive flow each time the residual network G_f is given. In order to make the algorithm efficient, we need to adopt a strategy that constructs a positive flow in the given residual network G_f effectively so that the number of executions of the **while** loop in step 3 is as small as possible. Many algorithms have been proposed for finding such a positive flow. In the next section, we describe an important technique, the *shortest path saturation* method, for constructing a positive flow given a flow network. We will see that when this method is adopted, the algorithm **Ford-Fulkerson** is efficient.

2.2 Shortest path saturation method

The method of *shortest path saturation* is among the most successful methods in constructing a positive flow in the residual network G_f to limit the number of executions of the **while** loop in step 3 of the algorithm **Ford-Fulkerson**, where the *length* of a path is measured by the number of edges in the path.

In the rest discussions in this chapter, we always assume that the flow network G has n vertices and m edges.

We first briefly describe an algorithm suggested by Edmond and Karp [35]. Edmond and Karp considered the method of constructing a positive flow for the residual network G_f by finding a shortest path from s to t in G_f and saturating it. Intuitively, each execution of this process saturates at least one edge from a shortest path from s to t in the residual network G_f. Thus, after $O(m)$ such executions, all shortest paths from s to t in G_f are saturated, and the distance from the source s to the sink t should be increased. This implies that after $O(nm)$ such executions, the distance from s to t should be larger than n, or equivalently, the sink t will become unreachable from the source s. Therefore, if we adopt Edmond and Karp's method to find positive flow in the residual network G_f, then the **while** loop in step 3 in the algorithm **Ford-Fulkerson** is executed at most $O(nm)$ times. Since a shortest path in the residual network G_f can be found in time $O(m)$ (using, for example, breadth first search), this concludes that Edmond-Karp's algorithm finds the maximum flow in time $O(nm^2)$.

Dinic [33] proposed a different approach. Instead of finding a single shortest path in the residual network G_f, Dinic finds *all* shortest paths from s to t in G_f, then saturates *all* of them. In the following, we give a detailed analysis for this approach.

Definition 2.2.1 Let G be a flow network. A flow f in G is a *shortest* *saturation flow* if (1) $f(u,v) > 0$ implies that $[u,v]$ is an edge in a shortest path from s to t in G, and (2) the flow f saturates every shortest path from s to t in G.

For each vertex v in a flow network G with source s and sink t, denote by $dist(v)$ the length of (i.e., the number of edges in) the shortest path in G from the source s to v (the *distance from s to v*). Similarly, if f is a flow in G, we let $dist_f(v)$ be the length of the shortest path from s to v in the residual network G_f.

Lemma 2.2.1 *Let G be a flow network with source s and sink t, and let f be a shortest saturation flow in G. Then $dist(t) < dist_f(t)$.*

PROOF. First we note that if a vertex v is on a shortest path P from the source s to a vertex w in G, then the subpath of P from s to v is a shortest path from s to v.

We prove two facts.

Fact 1: Suppose $[v,w]$ is an edge in G_f, then $dist(w) \le dist(v) + 1$.

Suppose $[v,w]$ is an edge in G. Then since any shortest path from s to v plus the edge $[v,w]$ is a path from s to w, whose length cannot be smaller than $dist(w)$, we have $dist(w) \le dist(v) + 1$.

If $[v,w]$ is not an edge in G, then since $[v,w]$ is an edge in the residual network G_f of G with respect to the flow f, the flow value $f(w,v)$ must be larger than 0. Since f is a shortest saturation flow, only for edges in shortest paths from s to t in G, f may have positive flow value. Thus $[w,v]$ must be an edge in a shortest path P from s to t in G. Then the subpath of P from s to w is a shortest path from s to w, and the subpath of P from s to v is a shortest path from s to v. That is, $dist(w) = dist(v) - 1$, which of course implies $dist(w) \le dist(v) + 1$.

Fact 2: For any vertex v, we have $dist(v) \le dist_f(v)$.

Suppose $r = dist_f(v)$. Let $(s, v_1, v_2, \ldots, v_{r-1}, v)$ be a shortest path in G_f from s to v. Then by Fact 1, we have $dist(v) \le dist(v_{r-1}) + 1$, $dist(v_i) \le dist(v_{i-1}) + 1$, for $i = 2, \ldots, r-1$, and $dist(v_1) \le dist(s) + 1$. Thus,

$$\begin{array}{rcl} dist(v) & \le & dist(v_{r-1}) + 1 \\ & \le & dist(v_{r-2}) + 2 \\ & \cdots & \\ & \le & dist(v_1) + (r-1) \\ & \le & dist(s) + r \end{array}$$

$$= \quad r \quad = \quad dist_f(v)$$

This proves Fact 2.

Now we are ready to prove our lemma.

Fact 2 shows that $dist(t) \leq dist_f(t)$. Hence, to prove the lemma, we only need to show that $dist(t)$ and $dist_f(t)$ are distinct. Let us assume the contrary that $dist(t) = dist_f(t) = r$ and derive a contradiction.

Let $P = (v_0, v_1, \ldots, v_{r-1}, v_r)$ be a shortest path in the residual network G_f from the source s to the sink t, where $v_0 = s$ and $v_r = t$. By Fact 1, we have

$$\begin{aligned} dist(v_r) &\leq dist(v_{r-1}) + 1 \\ &\leq dist(v_{r-2}) + 2 \\ &\cdots \\ &\leq dist(v_0) + r \\ &= dist(s) + r = r \end{aligned}$$

By our assumption, we also have $dist(v_r) = dist(t) = r$. Thus, all inequalities "$\leq$" in the above formula should be equality "$=$". This gives $dist(v_{i+1}) = dist(v_i) + 1$ for all $i = 0, \ldots, r-1$. But this implies that all $[v_i, v_{i+1}]$ are also edges in the original flow network G. In fact, if $[v_i, v_{i+1}]$ is not an edge in G, then since $[v_i, v_{i+1}]$ is an edge in the residual network G_f, $[v_{i+1}, v_i]$ must be an edge in G with the flow value $f(v_{i+1}, v_i) > 0$. Since f is a shortest saturation flow, $f(v_{i+1}, v_i) > 0$ implies that the edge $[v_{i+1}, v_i]$ is in a shortest path in G from s to t. But this would imply that $dist(v_{i+1}) = dist(v_i) - 1$, contradicting the fact $dist(v_{i+1}) = dist(v_i) + 1$.

Thus all $[v_i, v_{i+1}]$, $i = 0, \ldots, r-1$, are edges in the original flow network G, so P is also a path in G. Since P is of length r and $dist(t) = r$, P is a shortest path from s to t. Since f is a shortest saturation flow, the path P in G must be saturated by f, i.e., one of the edges in P is saturated by the flow f, which, by the definition of residual networks, should not appear in the residual network G_f. But this contradicts the assumption that P is a path in G_f.

This contradiction shows that we must have $dist(t) < dist_f(t)$. $\square$

Now we are ready to discuss how the shortest path saturation method is applied to the algorithm **Ford-Fulkerson**.

Theorem 2.2.2 *If in each execution of the* **while** *loop in step 3 in the algorithm* **Ford-Fulkerson***, we construct a shortest saturation flow f^* for* *the residual network G_f, then the number of executions of the* **while** *loop is bounded by $n-1$.*

PROOF. Suppose that f^* is a shortest saturation flow in the residual network G_f. By Lemma 2.2.1, the distance from s to t in the residual network $(G_f)_{f^*}$ (of G_f with respect to f^*) is at least 1 plus the distance from s to t in the original residual network G_f. Note that the residual network $(G_f)_{f^*}$ of G_f with respect to f^* is the residual network G_{f+f^*} of the original flow network G with respect to the new flow $f+f^*$. This can be easily verified by the following relation:

$$\begin{aligned} cap_f(u,v) - f^*(u,v) &= cap(u,v) - (f(u,v) + f^*(u,v)) \\ &= cap(u,v) - [f+f^*](u,v) \end{aligned}$$

Thus, $cap_f(u,v) > f^*(u,v)$ if and only if $cap(u,v) > [f+f^*](u,v)$, or equivalently, $[u,v]$ is an edge in $(G_f)_{f^*}$ if and only if it is an edge in G_{f+f^*}.

Therefore, the distance from s to t in the current residual network G_f is at least 1 plus the distance from s to t in the residual network G_f in the previous execution of the **while** loop. Since before the **while** loop, the distance from s to t in $G_f = G$ is at least 1 (the source s and the sink t are distinct in G), we conclude that after $n-1$ executions of the **while** loop, the distance from s to t in the residual network G_f is at least n. This means that the sink t is not reachable from the source s in the residual network G_f. By Theorem 2.1.4, the algorithm **Ford-Fulkerson** stops with a maximum flow f. $\Box$

The problem left is how a shortest saturation flow can be constructed for the residual network G_f. By the definition, a shortest saturation flow saturates all shortest paths from s to t and has positive value only on edges on shortest paths from s to t. Thus, constructing a shortest saturation flow can be split into two steps: (1) finding all shortest paths from s to t in G_f, and (2) saturating all these paths.

Since there can be too many (up to $\Omega(2^{cn})$ for some constant $c > 0$) shortest paths from s to t, it is infeasible to enumerate all of them. Instead, we construct a subnetwork L_0 in G_f, called the *layered network*, that contains exactly those edges contained in shortest paths of G_f from s to t.

The layered network L_0 of G_f can be constructed using a modification of the well-known *breadth first search* process, given in Figure 2.6, where Q is a queue that is a data structure serving for "first-in-first-out".

Stage 1 of the algorithm **Layered-Network** is a modification of the standard breadth first search process. The stage assigns a value $dist(v)$

```
Algorithm. Layered-Network

Input:  the residual network $G_f = (V_f, E_f)$
Output:  the layered network $L_0 = (V_0, E_0)$ of $G_f$

Stage 1. {constructing all shortest paths from $s$ to each vertex}
1. $V_0 = \emptyset$;   $E_0 = \emptyset$;
2. for all vertices $v$ in $G_f$ do   $dist(v) = \infty$;
3. $dist(s) = 0$;   $Q \leftarrow s$;
4. while $Q$ is not empty do
       $v \leftarrow Q$;
       for each edge $[v, w]$ do
           if $dist(w) = \infty$ then
               $Q \leftarrow w$;   $dist(w) = dist(v) + 1$;
               $V_0 = V_0 \cup \{w\}$;   $E_0 = E_0 \cup \{[v, w]\}$;
           else if $dist(w) = dist(v) + 1$ then  $E_0 = E_0 \cup \{[v, w]\}$;

Stage 2. {deleting vertices not in a shortest path from $s$ to $t$}
5.  let $L_0^r$ be $L_0 = (V_0, E_0)$ with all edge directions reversed;
6.  perform a breadth first search on $L_0^r$, starting from $t$;
7.  delete the vertices $v$ from $L_0$ if $v$ is not marked in step 6.
```

Figure 2.6: Construction of the layered network L_0

to each vertex v, which equals the distance from the source s to v, and includes an edge $[v, w]$ in L_0 only if $dist(v) = dist(w) - 1$. The difference of this stage from the standard breadth first search is that for an edge $[v, w]$ with $dist(v) = dist(w) - 1$, even if the vertex w has been in the queue Q, we still include the edge $[v, w]$ in L_0 to record the shortest paths from s to w that contain the edge $[v, w]$. Therefore, after stage 1, for each vertex v, exactly those edges contained in shortest paths from s to v are included in the network $L_0 = (V_0, E_0)$.

Stage 2 of the algorithm is to delete from L_0 all vertices (and their incident edges) that are not in shortest paths from the source s to the sink t. Since L_0 contains only shortest paths from s to each vertex and every vertex in L_0 is reachable from s in L_0, a vertex v is not contained in any shortest path from s to t if and only if t is not reachable from v in the network L_0, or equivalently, v is not reachable from t in the reversed network L_0^r. Step 6 in the algorithm identifies those vertices that are reachable from t in L_0^r, and step 7 deletes those vertices that are not identified in step 6.

Therefore, the algorithm **Layered-Network** correctly constructs the layered network L_0 of the residual network G_f. By the well-known analysis for the breadth first search process, the running time of the algorithm **Layered-Network** is bounded by $O(m)$.

Having obtained the layered network L_0, we now construct a shortest saturation flow so that for each path from s to t in L_0, at east one edge is saturated. There are two different methods for this, which are described in the following two subsections.

2.2.1 Dinic's algorithm

Given the layered network L_0, Dinic's algorithm for saturating all shortest paths from s to t in G_f is very simple, and can be described as follows. Starting from the vertex s, we follow the edges in L_0 to find a maximal path P of length at most $dist(t)$. Since the network L_0 is layered and contains only edges in the shortest paths from s to t in G_f, the path P can be found in a straightforward way (i.e., at each vertex, simply follow an arbitrary outgoing edge from the vertex). Thus, the path P can be constructed in time $O(dist(t)) = O(n)$. Now if the ending vertex is t, then we have found a path from s to t. We trace back the path P to find the edge e on P with minimum capacity c. Now we can push c amount of flow along the path P. Then we delete the edges on P that are saturated by the new flow. Note that this deletes at least one edge from the layered network L_0. On the other hand, if the ending vertex v of P is not t, then v must be a "deadend". Thus, we can delete the vertex v (and all incoming edges to v). In conclusion, in the above process of time $O(n)$, at least one edge is removed from the layered network L_0. Thus, after at most m such processes, the vertices s and t are disconnected, i.e., all shortest paths from s to t are saturated. This totally takes time $O(nm)$. A formal description for this process is given in Figure 2.7.

For completeness, we present in Figure 2.8 the complete Dinic's algorithm for constructing a maximum flow in a given flow network.

Theorem 2.2.3 *The running time of Dinic's maximum flow algorithm (Algorithm* **Max-Flow-Dinic** *in Figure 2.8) is* $O(n^2m)$.

PROOF. Theorem 2.2.2 claims that the **while** loop in step 3 in the algorithm is executed at most $n - 1$ times. In each execution of the loop, constructing the layered network L_0 by **Layered-Network** takes time $O(m)$. Constructing the shortest saturation flow f^* in G_f from the layered network L_0 by **Dinic-Saturation** takes time $O(nm)$. All other steps in the loop takes time at most $O(n^2)$. Therefore, the total running time for the

```
Algorithm.  Dinic-Saturation

Input:  the layered network $L_0$
Output:  a shortest saturation flow $f^*$ in $G_f$

1. while there is an edge from $s$ in $L_0$ do
       find a path $P$ of maximal length from $s$ in $L_0$;
       if $P$ leads to $t$
       then saturate $P$ and delete at least one edge on $P$;
       else delete the last vertex of $P$ from $L_0$.
```

Figure 2.7: Dinic's algorithm for a shortest saturation flow

algorithm **Max-Flow-Dinic** is bounded by $O(n^2m)$. □

2.2.2 Karzanov's algorithm

In Dinic's algorithm **Max-Flow-Dinic**, the computation time for each execution of the **while** loop in step 3 is dominated by the substep of constructing the shortest saturation flow f^* in G_f from the layered network L_0. Therefore, if this substep can be improved, then the time complexity of the whole algorithm can be improved. In this subsection, we show an algorithm by Karzanov [80] that improves this substep.

Let us have a closer look at our construction of the shortest saturation flow f^* in the algorithm **Max-Flow-Dinic**. With the layered network L_0 being constructed, we iterate the process of searching a path in L_0 from the source s to the sink t, pushing flow along the path, and saturating (thus cutting) at least one edge on the path. In the worst case, for each such a path, we may only be able to cut one edge. Therefore, to ensure that the source s is eventually separated from the sink t in L_0, we may have to perform the above iteration m times.

The basic idea of Karzanov's algorithm is to reduce the number of times we have to perform the above iteration from m to n. In each iteration, instead of saturating an edge in L_0, Karzanov saturates a vertex in L_0. Since there are at most n vertices in the layered network L_0, the number of iterations will be bounded by n.

Definition 2.2.2 Let v be a vertex in the layered network $L_0 = (V_0, E_0)$.

```
Algorithm.  Max-Flow-Dinic

Input:  a flow network $G$
Output:  a maximum flow $f$ in $G$

1. let $f(u,v) = 0$ for all pairs $(u,v)$ of vertices in $G$;
2. construct the residual network $G_f$;
3. while there is a positive flow in $G_f$ do
       call Layered-Network to construct the layered
           network $L_0$ for $G_f$;
       call Dinic-Saturation on $L_0$ to construct a shortest
           saturation flow $f^*$ in $G_f$;
       let $f = f + f^*$ be the new flow in $G$;
       construct the residual network $G_f$;
```

Figure 2.8: Dinic's algorithm for maximum flow

Define the *capacity*, *cap*(v), of the vertex v to be

$$cap(v) = \min\left(\sum_{[w,v]\in E_0} cap(w,v), \sum_{[v,u]\in E_0} cap(v,u)\right)$$

That is, *cap*(v) is the maximum amount of flow we can push through the vertex v. For the source s and the sink t, we naturally define

$$cap(s) = \sum_{[s,u]\in E_0} cap(s,u) \qquad \text{and} \qquad cap(t) = \sum_{[w,t]\in E_0} cap(w,t)$$

If we start from an arbitrary vertex v and try to push a flow of amount *cap*(v) through v, it may not always be possible. For example, pushing *cap*$(v) = 10$ units flow through a vertex v may require to push 5 units flow along an edge (v,w), which requires that *cap*(w) is at least 5. But the capacity of the vertex w may be less than 5, thus we would be blocked at the vertex w. However, if we always pick the vertex w in L_0 with the smallest capacity, this problem will disappear. In fact, trying to push a flow of amount *cap*(w), where w has the minimum *cap*(w), will require no more than *cap*(v) amount of flow to go through a vertex v for any vertex v. Therefore, we can always push the flow all the way to the sink t (assuming we have no deadend vertices). Similarly, we can *pull* this amount *cap*(w) of flow from the incoming edges of w all the way back to the source s. Note

```
Algorithm.  Karzanov-Initiation

Input:  the layered network $L_0$
Output:  the vertex capacity for each vertex

1. for each vertex $v \neq s, t$ do    $in[v] = 0$; $out[v] = 0$;
2. $in[s] = +\infty$; $out[t] = +\infty$;
3. for each edge $[u, v]$ in $L_0$ do
       $in[v] = in[v] + cap(u, v)$;    $out[u] = out[u] + cap(u, v)$;
4. for each vertex $v$ in $L_0$ do    $cap[v] = \min\{in[v], out[v]\}$.
```

Figure 2.9: Computing the capacity for each vertex

that this process saturates the vertex w. Thus, the vertex w can be removed from the layered network L_0 in the remaining iterations.

Now we can formally describe Karzanov's algorithm. The first subroutine given in Figure 2.9 computes the capacity for each vertex in the layered network L_0.

We will always start with a vertex v with the smallest $cap(v)$ and push a flow f^v of amount $cap(v)$ through it all the way to the sink t. This process, called **Push**(v, f^v) and given in Figure 2.10, is similar to the breadth first search process, starting from the vertex v. We use the array $fl[w]$ to record the amount of flow requested to push through the vertex w, and $fl[w] = 0$ implies that the vertex w has not been seen in the breadth first search process.

We make a few remarks on the algorithm **Push**(v, f^v). First we assume that there is no dead-vertex in the layered network L_0. That is, every edge in L_0 is on a shortest path from s to t. This condition holds when the layered network L_0 is built by the algorithm **Layered-Network**. We will keep this condition in Karzanov's main algorithm when vertices and edges are deleted from L_0.

The algorithm **Push**(v, f^v), unlike the standard breadth first search, may not search all vertices reachable from v. Instead, for each vertex u, with a requested flow amount $fl[u]$ through it, the algorithm looks at the out-going edges from u to push the flow of amount $fl[u]$ through these edges. Once this amount of flow is pushed through some of these edges, the algorithm will ignore the rest of the out-going edges from u. Note that since the vertex v has the minimum $cap(v)$, and no $fl[u]$ for any vertex u is larger than $cap(v)$, the amount $fl[u]$ can never be larger than $cap(u)$. Thus, the

```
Algorithm.  Push($v, f^v$)
Input:  the layered network $L_0$
1.      $Q \leftarrow v$; {$Q$ is a queue}   $fl[v] = cap(v)$;
2.      while $Q$ is not empty do
2.1.        $u \leftarrow Q$;   $f_0 = fl[u]$;
2.2.        while $f_0 > 0$ do
                pick an out-going edge $[u, w]$ from $u$;
                if $fl[w] = 0$ and $w \neq t$ then $Q \leftarrow w$;
2.2.1.          if $cap(u, w) \leq f_0$ then
                    $f^v(u, w) = cap(u, w)$;   delete the edge $[u, w]$;
                    $fl[w] = fl[w] + cap(u, w)$;   $f_0 = f_0 - cap(u, w)$;
2.2.2.          else {$cap(u, w) > f_0$}
                    $f^v(u, w) = f_0$;   $fl[w] = fl[w] + f_0$;
                    $cap(u, w) = cap(u, w) - f_0$;   $f_0 = 0$;
2.3.        if $u \neq v$ then $cap(u) = cap(u) - fl[u]$;
2.4.        if $u \neq v$ and $cap(u) = 0$ then delete the vertex $u$.
```

Figure 2.10: Pushing a flow of value $cap(v)$ from v to t

amount $fl[u]$ of flow can always be pushed through the vertex u.

When a flow f' is pushed along an edge $[u, w]$, we add f' to $fl[w]$ to record this new requested flow through the vertex w. Note that when a vertex u is picked from the queue Q in step 2.1, the flow requested along in-coming edges to u has been completely decided. Thus, $fl[u]$ records the correct amount of flow to be pushed through u.

Also note that in the subroutine **Push**(v, f^v), we neither change the value $cap(v)$ nor remove the vertex v from the layered network L_0. This is because the vertex v will be used again in the following **Pull**(v, f^v) algorithm.

The algorithm **Pull**(v, f^v) is very similar to algorithm **Push**(v, f^v). We start from the vertex v and pull a flow f^v of amount $cap(v)$ all the way back to the source vertex s. Note that now the breadth first search is on the reversed directions of the edges of the layered network L_0. Thus, we will also keep a copy of the reversed layered network L_0^r, which is L_0 with all edge directions reversed. The reversed layered network L_0^r can be constructed from the layered network L_0 in time $O(m)$ (this only needs to be done once for all calls to **Pull**). Moreover, note that the only vertex that can be seen in

```
Algorithm.  Pull($v, f^v$)
Input:  the reversed layered network $L_0^r$
1.      $Q \leftarrow v$; {$Q$ is a queue}   $fl[v] = cap(v)$;
2.      while $Q$ is not empty do
2.1.        $u \leftarrow Q$;   $f_0 = fl[u]$;
2.2.        while $f_0 > 0$ do
                pick an in-coming edge $[w, u]$ to $u$;
                if $fl[w] = 0$ and $w \neq s$ then $Q \leftarrow w$;
2.2.1.          if $cap(w, u) \leq f_0$ then
                    $f^v(w, u) = cap(w, u)$;   delete the edge $[w, u]$;
                    $fl[w] = fl[w] + cap(w, u)$;   $f_0 = f_0 - cap(w, u)$;
2.2.2.          else {$cap(w, u) > f_0$}
                    $f^v(w, u) = f_0$;   $fl[w] = fl[w] + f_0$;
                    $cap(w, u) = cap(w, u) - f_0$;   $f_0 = 0$;
2.3.        $cap(u) = cap(u) - fl[u]$;
2.4.        if $cap(u) = 0$ then delete the vertex $u$.
```

Figure 2.11: Pulling a flow of value $cap(v)$ from s to v

both algorithms **Push**(v, f^v) and **Pull**(v, f^v) is the vertex v — **Push**(v, f^v) is working on the vertices "after" v in L_0 while **Pull**(v, f^v) is working on the vertices "before" v in L_0. Therefore, no updating is needed between the call to **Push**(v, f^v) and the call to **Pull**(v, f^v). The algorithm **Pull**(v, f^v) is given in Figure 2.11.

After the execution of the **Pull**(v, f^v) algorithm, the vertex v with minimum capacity always gets removed from the layered network L_0.

With the subroutines **Push** and **Pull**, Karzanov's algorithm for constructing a shortest saturation flow f^* in the residual network G_f is given in Figure 2.12.

Some implementation details should be explained for the algorithm **Karzanov-Saturation**.

The dynamic deletions of edges and vertices from in the layered network L_0 can be recorded by the two dimensional array $cap(\cdot, \cdot)$: we set $cap(u, w) = 0$ when the edge $[u, w]$ is deleted from L_0. The actual deletion of the item $[u, w]$ from the adjacency list representation of the layered network L_0 or of the reversed layered network L_0^r is done later: when we scan the adjacency list and encounter an item $[u, w]$, we first check if $cap(u, w) = 0$. If so, we

```
Algorithm. Karzanov-Saturation
Input: the layered network $L_0$
Output: a shortest saturation flow $f^*$ in $G_f$

1. call Karzanov-Initiation to compute the vertex capacity $cap(v)$
       for each vertex $v$;
2. $f^* = 0$;
3. while there is a vertex in $L_0$ do
       pick a vertex $v$ in $L_0$ with minimum $cap(v)$;
       if $v$ is a dead-vertex then   delete $v$ from $L_0$
       else  call Push$(v, f^v)$;  call Pull$(v, f^v)$;  $f^* = f^* + f^v$
```

Figure 2.12: Karzanov's algorithm for shortest saturation flow

delete the item (using an extra $O(1)$ time) and move to the next item in the list.

Similarly, we keep a vertex array to record whether a vertex is in the layered network L_0. There are two ways to make a vertex v become a dead-vertex: (1) $cap(v) = 0$, which means either all in-coming edges to v or all out-going edges from v are saturated; and (2) v becomes a dead-vertex because of removal of other dead-vertices. For example, suppose that v has only one in-coming edge $[u, v]$ of capacity 10 and one out-going edge $[v, w]$ of capacity 5. Then $cap(v) = 5 \neq 0$. However, if w becomes a dead-vertex (e.g., because all out-going edges from w are saturated) and is deleted, then the vertex v will also become a dead-vertex. For this, we also record the number $in[v]$ of in-coming edges and the number $out[v]$ of out-going edges for each vertex v. Once one of $in[v]$ and $out[v]$ becomes 0, we set $cap(v) = 0$ immediately. Since in step 3 of the algorithm **Karzanov-Saturation** we always pick the vertex of minimum $cap(v)$, dead-vertices in L_0 will always be picked first. Consequently, when the subroutines **Push** and **Pull** are called in step 3, there must be no dead-vertex in the current layered network L_0.

We now analyze the algorithm **Karzanov-Saturation**.

Lemma 2.2.4 *The algorithm* **Karzanov-Saturation** *takes time* $O(n^2)$.

PROOF. Step 1 takes time $O(e) = O(n^2)$. Since each execution of the **while** loop deletes at least one vertex v from L_0 (either because v is a dead-vertex or because of the subroutine call **Pull**(v, f^v)), the **while** loop in step 3 is executed at most n times.

The first substep in step 3, finding a vertex v of minimum $cap(v)$ in the layered network L_0, takes time $O(n)$. Thus, the total time spent by the algorithm **Karzanov-Saturation** on this substep is bounded by $O(n^2)$.

The analysis for the time spent on the subroutine calls to **Push** and **Pull** is a bit more tricky. Let us first consider the subroutine **Push**(v, f^v). To push a flow of amount $fl[u]$ through a vertex u, we take each out-going edge from u. If the capacity of the edge is not larger than the amount of flow we request to push (step 2.2.1 in the algorithm **Push**(v, f^v)), we saturate and delete the edge; if the capacity of the edge is larger than the amount of flow we request to push (step 2.2.2 in the algorithm **Push**(v, f^v)), we let all remaining flow go along that edge and jump out from the **while** loop in Step 2.2 in the algorithm **Push**(v, f^v). Moreover, once an edge gets deleted in the algorithm, the edge will never appear in the layered network L_0 for the later calls for **Push** in the algorithm **Karzanov-Saturation**. Thus, each execution of the **while** loop in step 2.2 of the algorithm **Push**(v, f^v), except maybe the last one, deletes an edge from the layered network L_0. Hence, the total number of such executions in the whole algorithm **Karzanov-Saturation** cannot be larger than m plus n times the number of calls to **Push**, where m is for the executions of the loop that delete an edge in L_0, and n is for the executions of the loop that do not delete edges. Therefore, the total number of executions of the **while** loop in step 2.2 in the algorithm **Push**(v, f^v) for all calls to **Push** in the algorithm **Karzanov-Saturation** is bounded by $O(n^2)$. Since each execution of this **while** loop takes constant time and this part dominates the running time of the algorithm **Push**, we conclude that the total time spent by **Karzanov-Saturation** on the calls to **Push** is bounded by $O(n^2)$. Similarly, the total time spent on the calls to the subroutine **Pull** is also bounded by $O(n^2)$. $\square$

Now if we replace the call to **Dinic-Saturation** in the algorithm **Max-Flow-Dinic** by a call to **Karzanov-Saturation**, we get Karzanov's maximum flow algorithm, which is given in Figure 2.13.

By Theorem 2.2.2, Lemma 2.2.4, and the analysis for the algorithm **Layered-Network**, we get immediately,

Theorem 2.2.5 *Karzanov's maximum flow algorithm (algorithm* **Max-Flow-Karzanov***) runs in time* $O(n^3)$

```
**Algorithm. Max-Flow-Karzanov**
Input: a flow network $G$
Output: a maximum flow $f$ in $G$

1. let $f(u,v) = 0$ for all pairs $(u,v)$ of vertices in $G$;
2. construct the residual network $G_f$;
3. **while** there is a positive flow in $G_f$ **do**
       call **Layered-Network** to construct the layered
           network $L_0$ for $G_f$;
       call **Karzanov-Saturation** on $L_0$ to construct a
           shortest saturation flow $f^*$ in $G_f$;
       let $f = f + f^*$ be the new flow in $G$;
       construct the residual network $G_f$;
```

Figure 2.13: Karzanov's algorithm for maximum flow

2.3 Preflow method

The preflow method was proposed more recently by Goldberg and Tarjan [55]. To describe this method, let us start by reviewing Karzanov's maximum flow algorithm. Consider the subroutine **Push**(v, f^v) in Karzanov's algorithm. On each vertex u in the search, we try to push the requested amount $fl[u]$ of flow through the vertex u. Since the operation uses only the local neighborhood relation for the vertex u and is independent of the global structure of the underlying flow network, the operation is very efficient. On the other hand, Karzanov's algorithm seems a bit too conservative: it pushes a flow of value $fl[u]$ through the vertex u only when it knows that this amount of flow can be pushed all the way to the sink t. Moreover, it pushes flow only along the shortest paths from s to t. Can we generalize this efficient operation so that a larger amount of flow can be pushed through each vertex along all possible paths (not just shortest paths) to the sink t?

Think of the flow network as a system of water pipes, in which vertices correspond to pipe junctions. Each junction has a position such that water only flows from higher positions to lower positions. In particular, the position of the sink is the lowest, and the position of the source is always higher than that of the sink. For each junction u, we have a requested amount $e[u]$ of flow to be pushed through the junction, which at the beginning is supposed to be stored in a private reservoir for the junction u. Now if there is a non-saturated pipe $[u, w]$ such that the position of w is lower than u, then a certain amount of flow can be pushed along the pipe $[u, w]$. The pushed flow seems to flow to the sink since the sink has the lowest position. In case no further push is possible and there are still junctions with non-empty reservoir, we "lift" the positions for the junctions with non-empty reservoir to make further pushes possible from these junctions.

How do we decide the requested flow $e[u]$ for each junction? According to the principal "higher pressure induces higher speed," we try to be a bit aggressive, and let $e[u]$ be the amount requested from the incoming pipes to u, which may be larger than the capacity of u. It may eventually turn out that this request is too much for the junction u to let through. In this case, we observe that with the position of the junction u increased, eventually the position of u is higher than the source. Thus, the excess flow $e[u]$ will go backward in the flow network all the way back to the source.

Let us formulate the above idea using the terminologies in the MAXIMUM FLOW problem.

Definition 2.3.1 Let $G = (V, E)$ be a flow network with source s and sink t. A function f on vertex pairs of G is a *preflow* if f satisfies the capacity constraint property, the skew symmetry property (see Definition 2.1.2), and the following *nonnegative excess property*: $\sum_{v \in V} f(v, w) \geq 0$ for all $w \in V - \{s\}$. The amount $\sum_{v \in V} f(v, w)$ is called the *excess flow* into the vertex w, denoted $e[w]$.

The excess flow $e[w]$ is the amount of further flow we want to push through the vertex w.

The concept of the residual network can be extended to preflows in a flow network. Formally, suppose that f is a preflow in a flow network G, then the *residual network* G_f (with respect to f) has the same vertex set as G, and $[u, v]$ is an edge of capacity $cap_f(u, v) = cap(u, v) - f(u, v)$ in G_f if and only if $cap(u, v) > f(u, v)$.

Note that both the processes described above of pushing along non-saturated edges and of sending excess flow back to the source can be implemented by a single type of push operation on edges in the residual network: if an edge $[u, v]$ is non-saturated then the edge $[u, v]$ also exists in the residual network, and if there is a positive flow request from s to u along a path that should be sent back to the source, then the reversed path from u to s is a path in the residual network.

Each flow network G is also associated with a *height function* h such that for any vertex u of G, $h(u)$ is a non-negative integer. To facilitate the analysis of our algorithms, we require that the height function be more

```
Algorithm. Push$(u, w)$
{ Applied only when $(f, h)$ is a preflow scheme, $h(u) = h(w) + 1$,
      $cap_f(u, w) > 0$, and $e[u] > 0$. }
1.  $f_0 = \min\{e[u], cap_f(u, w)\}$;
2.  $f(u, w) = f(u, w) + f_0$;  $f(w, u) = -f(u, w)$;
3.  $e[w] = e[w] + f_0$;   $e[u] = e[u] - f_0$.
```

Figure 2.14: Pushing a flow along the edge $[u, w]$

restricted when it is associated with a preflow, as given in the following definition.

Definition 2.3.2 Let G be a flow network, f be a preflow in G, and h be a height function for G. The pair (f, h) is a *preflow scheme* for G if for any edge $[u, w]$ in the residual network G_f, we have $h(u) \leq h(w) + 1$.

Now we are ready to describe our first basic operation on a flow network with a preflow scheme.

The operation **Push**(u, w) is applied to a pair of vertices u and w in a flow network G only when the following three conditions all holds: (1) $h(u) = h(w) + 1$; (2) $cap_f(u, w) > 0$; and (3) $e[u] > 0$. In this case, the **Push** operation pushes as much flow as possible (i.e., the minimum of $cap_f(u, w)$ and $e[u]$) along the edge $[u, w]$, and update the values for $e[u]$, $e[w]$, and $f(u, w)$. In other words, the operation shifts an amount of $\min\{cap_f(u, w), e[u]\}$ excess value, along edge $[u, w]$, from u to w. A formal description of the operation **Push**(u, w) is given in Figure 2.14.

Note that the operation **Push**(u, w) does not change the height function value. On the other hand, the operation does change the flow value $f(u, w)$, and the excess values of the vertices u and w. The following lemma shows that the operation **Push** preserves a preflow scheme.

Lemma 2.3.1 *Let (f, h) be a preflow scheme for a flow network G. Suppose that the operation* **Push**(u, w) *is applicable to a pair of vertices u and w in G. Then after applying the operation* **Push**(u, w)*, the new values for f and h still make a preflow scheme.*

PROOF. The **Push**(u, w) operation only changes values related to vertices u and w and to edge $[u, w]$. For the new value for f, (1) the capacity

```
Algorithm. Lift($v$)
{ Applied only when $(f, h)$ is a preflow scheme, $e[v] > 0$, and for
    all out-going edges $[v, w]$ from $v$ (there is at least one such
    an edge), $h(v) < h(w) + 1$. }
1.  let $w_0$ be a vertex with the minimum $h(w_0)$ over all $w$
        such that $[v, w]$ is an edge in $G_f$;
2.  $h(v) = h(w_0) + 1$;
```

Figure 2.15: Lifting the position of a vertex v

constraint is preserved: since $cap_f(u, w) \geq f_0$ and $cap_f(u, w)$ is equal to $cap(u, w)$ minus the old flow value $f(u, w)$, thus, $cap(u, w)$ is not smaller than the old flow value $f(u, w)$ plus f_0, which is the new flow value $f(u, w)$; (2) the skew symmetry property is preserved by step 2; and (3) the non-negative excess property is preserved: the excess $e[u]$ of u is decreased by $f_0 \leq e[u]$ and the excess $e[w]$ is in fact increased.

To consider the constraint for the height function, note that the only possible new edge that is created by the **Push**(u, w) operation is edge $[w, u]$. Since the **Push** operation does not change the height function values, we have $h[w] = h[u] - 1$, which is of course not larger than $h[u] + 1$. □

Now we consider the second basic operation, **Lift** on a preflow scheme. The **Lift** operation is applied to a vertex v in a preflow scheme (f, h) when the position of v is too low for the **Push** operation to push a flow through v. Therefore, to apply a **Lift** operation on a vertex v, the following three conditions should all hold: (1) $e[v] > 0$; (2) there is an out-going edge $[v, w]$ from v in the residual network G_f; and (3) for each out-going edge $[v, w]$ from v in G_f, we have $h(v) < h(w) + 1$ (note that condition (3) is equivalent to $h(v) \neq h(w) + 1$ since for a preflow scheme (f, h), we always have $h(v) \leq h(w) + 1$). The formal description of the **Lift** operation is given in Figure 2.15.

Note that the operation **Lift**(v) does not change the preflow value.

Lemma 2.3.2 *Let (f, h) be a preflow scheme for a flow network G. Suppose that* **Lift**(v) *is applicable to a vertex v in G. Then after applying the operation* **Lift**(v)*, the new values for f and h still make a preflow scheme.*

PROOF. Since the operation **Lift**(v) does not change the preflow value, we

```
Algorithm. Max-Flow-GT
Input: a flow network $G$ with source $s$ and sink $t$
Output: a maximum flow $f$ in $G$

1.   for each vertex $v$ in $G$ do $h(v) = 0$;  $e[v] = 0$;
2.   for each pair of vertices $u$ and $w$ in $G$ do $f(u, w) = 0$;
3.   $h(s) = n$;
4.   for each out-going edge $[s, v]$ from $s$ do
         $f(s, v) = -f(v, s) = cap(s, v)$;  $e[v] = cap(s, v)$;
5.   while there is a vertex $v \neq s, t$ with $e[v] > 0$ do
5.1.     pick a vertex $v \neq s, t$ with $e[v] > 0$;
5.2.     if Push is applicable to an edge $[v, w]$ in $G_f$
             then Push$(v, w)$ else Lift$(v)$.
```

Figure 2.16: Golberg-Tarjan's algorithm for maximum flow

only need to verify that the new values for the height function h still make a preflow scheme with the preflow f. For this, we only need to verify the edges in the residual network G_f that have the vertex v as an end.

For any in-coming edge $[u, v]$ to v, before the operation **Lift**(v), we have $h(u) \leq h(v) + 1$. Since the **Lift**(v) increases the height $h(v)$ of v by at least 1, we still have $h(u) \leq h(v) + 1$ after the **Lift**(v) operation.

For each out-going edge $[v, w]$ from v, by the choice of the vertex w_0, we have $h(w) \geq h(w_0)$. Thus, the new value of $h(v) = h(w_0) + 1$ is not larger than $h(w) + 1$. $\square$

Now we are ready to describe our maximum flow algorithm using the preflow method. The algorithm was developed by Goldberg and Tarjan [55]. The algorithm is given in Figure 2.16.

In the rest of this section, we first prove that the algorithm **Max-flow-GT** correctly constructs a maximum flow given a flow network, then we analyze the algorithm. Further improvement on the algorithm will also be briefly described.

Lemma 2.3.3 *Let f be a preflow in a flow network G. If a vertex u_0 has a positive excess $e[u_0] > 0$, then there is a path in the residual network G_f from u_0 to the source s.*

PROOF. Let V_0 be the set of vertices reachable from u_0 in the residual network G_f. Consider any pair of vertices v and w such that $v \in V_0$ and $w \notin V_0$. Since in G_f, the vertex v is reachable from the vertex u_0 while the vertex w is not reachable from u_0, there is no edge from v to w in G_f. Thus, $cap_f(v, w) = 0$, which by the definition implies that in the original flow network G we have $f(v, w) = cap(v, w) \geq 0$, $f(w, v) \leq 0$. Therefore, for any $v \in V_0$ and $w \notin V_0$, we must have $f(w, v) \leq 0$.

Now consider

$$\sum_{v \in V_0} e[v] = \sum_{v \in V_0} \sum_{w \in V} f(w, v) = \sum_{v \in V_0} \sum_{w \in V_0} f(w, v) + \sum_{v \in V_0} \sum_{w \notin V_0} f(w, v)$$

The first term in the right hand side is equal to 0 by the skew symmetry property, and the second term in the right hand side is not larger than 0 since for any $v \in V_0$ and $w \notin V_0$ we have $f(w, v) \leq 0$. Thus, we have

$$\sum_{v \in V_0} e[v] \leq 0 \tag{2.2}$$

Now if the source s is not in the set V_0, then since $e[u_0] > 0$ and by the non-negative excess property, for all other vertices v in V_0, we have $e[v] \geq 0$, we derive $\sum_{v \in V_0} e[v] > 0$, contradicting to the relation in (2.2).

In conclusion, the source s must by reachable from the vertex u_0, i.e., there must be a path from u_0 to s in the residual network G_f. □

Now we are ready to prove the correctness for the algorithm **Max-flow-GT**.

Lemma 2.3.4 *Goldberg-Tarjan's maximum flow algorithm (the algorithm* **Max-Flow-GT** *in Figure 2.16) stops with a maximum flow f in the given flow network G.*

PROOF. Steps 1-4 initialize the values for the functions f and h. It is easy to verify that after these steps, f is a preflow in the flow network G. To verify that at this point (f, h) is a preflow scheme, note that f has positive value only on the out-going edges from the source s, and f saturates all these egdes. Thus, in the residual network G_f, the source s has no out-going edges. Moreover, all vertices have height 0 except the source s, which has height n. Therefore, if $[u, v]$ is an edge in the residual network G_f, then $u \neq s$ and $h(u) = 0$. In consequence, for any edge $[u, v]$ in the residual network G_f, we must have $h(u) \leq h(v) + 1$. This shows that at the end of step 4 in the algorithm, (f, h) is a preflow scheme for the flow network G.

An execution of the **while** loop in step 5 applies either a **Push** or a **Lift** operation. By Lemma 2.3.1 and Lemma 2.3.2, if (f, h) is a preflow scheme for G before the operations, then the new values for f and h after these operations still make a preflow scheme. Inductively, before each execution of the **while** loop in step 5, the values of f and h make (f, h) a preflow scheme for the flow network G.

We must show the validity for step 5.2, i.e., if the operation **Push** does not apply to any out-going edge $[v, w]$ from the vertex v, then the operation **Lift** must apply. By step 5.1, we have $e[v] > 0$. By Lemma 2.3.3, there must be an outgoing edge $[v, w]$ from v in the residual network G_f. Therefore, if the operation **Push** does not apply to any out-going edge $[v, w]$ from v, then we must have $h(v) \neq h(w)+1$ for any such an edge. Since (f, h) is a preflow scheme, $h(v) \leq h(w) + 1$. Thus, $h(v) \neq h(w) + 1$ implies $h(v) < h(w) + 1$ for all out-going edges $[v, w]$ from v. Now this condition ensures that the **Lift**(v) operation must apply.

Now we prove that when the algorithm **Max-Flow-GT** stops, the resulting preflow f is a maximum (regular) flow in the flow network G. The algorithm stops when the condition in step 5 fails. That is, for *all* vertices $v \neq s, t$, we have $e[v] = 0$. By the definition, this means that for all vertices $v \neq s, t$, we have $\sum_{w \in V} f(w, v) = 0$. Thus, the function f satisfies the flow conservation property. Since f is a preflow, it also satisfies the capacity constraint property and the skew symmetry property. Thus, when the algorithm stops, the resulting preflow f is actually a regular flow in G.

To prove that this flow f is maximum, by Theorem 2.1.4, we only have to prove that there is no path from the source s to the sink t in the residual network. Suppose the opposite that there is a path P from s to t in the residual network G_f. Without loss of generality, we assume that the path P is a simple path of length less than n. Let $P = \{v_0, v_1, \ldots, v_k\}$, where $v_0 = s$, $v_k = t$ and $k < n$. Since $[v_i, v_{i+1}]$, for $0 \leq i \leq k-1$, are edges in G_f and the pair (f, h) is a preflow scheme for G, we must have $h(v_i) \leq h(v_{i+1}) + 1$ for all $1 \leq i \leq k-1$. Therefore,

$$\begin{aligned} h(s) = h(v_0) &\leq h(v_1) + 1 \\ &\leq h(v_2) + 2 \\ &\quad \cdots\cdots \\ &\leq h(v_k) + k \\ &= h(t) + k \end{aligned}$$

Since $h(s) = n$ and $h(t) = 0$ the above formula implies $n \leq k$ contradicting the assumption $n > k$. Therefore, there is no path from the source s to the sink t in the residual network G_f. Consequently, the flow f obtained by the algorithm is a maximum flow in the flow network G. $\square$

Now we analyze the algorithm **Max-Flow-GT**. The running time of the algorithm is dominated by step 5, which is in turn dominated by the number of subroutine calls to the operations **Push** and **Lift**. Thus, a bound on the subroutine calls will imply a bound to the running time of the algorithm.

Lemma 2.3.5 *Let (f, h) be the final preflow scheme obtained by the algorithm* **Max-Flow-GT** *for a given flow network G. Then for any vertex v_0 of G, we have $h(v_0) \leq 2n - 1$.*

PROOF. If the vertex v_0 never gets a positive excess $e[v] > 0$, then it is never picked up in step 5.1. Thus, the height of v is never changed. Since the initial height of each vertex in G is at most n, the lemma holds.

Now suppose that the vertex v_0 does get a positive excess $e[v_0] > 0$ during the execution of the algorithm. Let (f', h') be the last preflow scheme during the execution of the **while** loop in step 5 in which $e[v_0] > 0$. By Lemma 2.3.3, there is a path from v_0 to s in the residual network $G_{f'}$. Let $P' = \{v_0, v_1, \ldots, v_k\}$ be a simple path in $G_{f'}$ from v_0 to s, where $v_k = s$ and $k \leq n-1$. Then since (f', h') is a preflow scheme, we have $h'(v_i) \leq h'(v_{i+1}) + 1$, for all $0 \leq i \leq k-1$. This implies immediately $h'(v_0) \leq h'(s) + k \leq 2n-1$, since the height of the source s is n and is never changed.

Since the execution of the **while** loop on the preflow scheme (f', h') changes the excess $e[v_0]$ on v_0 from a positive value to 0, this execution must be a **Push** on an out-going edge from v_0, which does not change the height value for v_0. After this execution, since $e[v_0]$ remains 0 so v_0 is never picked by step 5.1 and its height value is never changed. In consequence, at the end of the algorithm, the height value $h(v_0)$ for the vertex v_0 is still not larger than $2n - 1$. $\square$

Now we can derive a bound on the number of calls to the subroutine **Lift** by the algorithm **Max-Flow-GT**.

Lemma 2.3.6 *The total number of calls to the subroutine* **Lift** *by the algorithm* **Max-Flow-GT** *is bounded by $2n^2 - 8$.*

PROOF. By Lemma 2.3.5, the height of a vertex v cannot be larger than $2n - 1$. Since each call **Lift**(v) increases the height of v by at least 1, the number of calls **Lift**(v) on the vertex v cannot be larger than $2n - 1$. Note that the operation **Lift** does not apply on the source s and the sink t. Thus, the total number of calls to **Lift** in the algorithm **Max-Flow-GT** cannot be larger than $(2n-1)(n-2) \leq 2n^2 - 8$ (note $n \geq 2$). $\square$

Lemma 2.3.5 can also be used to bound the number of calls to the subroutine **Push**.

Lemma 2.3.7 *The total number of calls to the subroutine* **Push** *by the algorithm* **Max-Flow-GT** *is bounded by* $O(n^2 m)$.

PROOF. A subroutine call to $\textbf{Push}(u, w)$ is a *saturating push* if it makes $f(u, w) = cap(u, w)$. Otherwise, the subroutine call is called a *non-saturating push*.

We first count the number of saturating pushes.

Suppose that we have a saturating push $\textbf{Push}(u, w)$ along the edge $[u, w]$ in the residual network G_f. Let the value of $h(u)$ at this moment be h_0. After this push, there is no edge from u to w in the residual network G_f. When can the next saturating push $\textbf{Push}(u, w)$ from u to w happen again? To make it possible, we first have to make $[u, w]$ an edge again in the residual network G_f. The only way to make $[u, w]$ an edge in the residual netowrk is to push a positive flow from w to u, i.e., a call $\textbf{Push}(w, u)$ must apply. In order to apply $\textbf{Push}(w, u)$, the height of vertex w must be larger than the height of vertex u. Therefore, the new value of $h(w)$ must be at least $h_0 + 1$. Now after the call $\textbf{Push}(w, u)$, a new edge $[u, w]$ is created in the residual network. Now similarly, if we want to apply another call to $\textbf{Push}(u, w)$ to the edge $[u, w]$ (no matter if it is saturating on non-saturating), the height $h(u)$ must be larger than the new height $h(w)$ of w. That is, the new height $h(u)$ is at lest $h_0 + 2$. Therefore, between two consecutive saturating pushes $\textbf{Push}(u, w)$ along the edge $[u, w]$ in the residual network G_f, the height of the vertex u is increased by at least 2. By Lemma 2.3.5, the height of the vertex u is bounded $2n - 1$. Thus, the total number of saturating pushes $\textbf{Push}(u, w)$ for the pair of vertices u and w is bounded by $(2n-1)/2 + 1 < n + 1$. Now note that a push $\textbf{Push}(u, w)$ applies only when $[u, w]$ is an edge in the residual network G_f, which implies that either $[u, w]$ or $[w, u]$ is an edge in the original network G. Thus, there are at most $2m$ pairs of vertices u and w on which a saturating push $\textbf{Push}(u, w)$ can apply. In summary, the total number of saturating pushes in the algorithm **Max-Flow-GT** is bounded by $2m(n + 1)$.

Now we count the number of non-saturating pushes.

Let V_+ be the set of vertices u in $V - \{s, t\}$ such that $e[u] > 0$. Consider the value $\beta_+ = \sum_{u \in V_+} h(u)$. The value β_+ is 0 before step 5 of the algorithm

Max-Flow-GT starts since at that point all vertices except s have height 0. The value β_+ is again 0 at the end of the algorithm since at this point all vertices $u \neq s, t$ have $e[u] = 0$. Moreover, the value β_+ can never be negative during the execution of the algorithm. Now we consider how each of the operations **Push** and **Lift** affects the value β_+.

If **Push**(u, w) is a non-saturating push, then before the operation $u \in V_+$ and $h(u) = h(w) + 1$, and after the operation, the excess $e[u]$ becomes 0 so the vertex u is removed from the set V_+. Note that the vertex w may be a new vertex added to the set V_+. Thus, the operation subtracts a value $h(u)$ from β_+, and *may* add a value $h(w) = h(u) - 1$ to β_+. In any case, the non-saturating push **Push**(u, w) decreases the value β_+ by at least 1.

If **Push**(u, w) is a saturating push, then u belongs to the set V_+ before the operation but w may be added to the set V_+ by the operation. By Lemma 2.3.5, the height $h(w)$ of w is bounded by $2n - 1$. Thus, each saturating push increases the value β_+ by at most $2n - 1$.

Now consider the **Lift**(v) operation. When the operation **Lift**(v) applies, the vertex v is in the set V_+. Since the height $h(v)$ of v cannot be larger than $2n - 1$, the operation **Lift**(v) increases the value β_+ by at most $2n - 1$. In consequence, the operation **Lift**(v) increases the value β_+ by at most $2n - 1$.

By Lemma 2.3.6, the total number of calls to the subroutine **Lift** is bounded by $2n^2 - 8$, and by the first part in this proof, the total number of saturating pushes is bounded by $2m(n + 1)$. Thus, the total value of β_+ increased by the calls to **Lift** and by the calls to saturating pushes is at most

$$(2n - 1)(2n^2 - 8 + 2m(n + 1)) \leq 4n^3 + 6n^2m$$

Since each non-saturating push decreases the value β_+ by at least 1 and the value β_+ is never less than 0, we conclude that the total number of non-saturating pushes by the algorithm is bounded by $4n^3 + 6n^2m = O(n^2m)$.

This completes the proof for the lemma. □

Now we conclude the discussion for the algorithm **Max-Flow-GT**.

Theorem 2.3.8 *Goldberg and Tarjan's maximum flow algorithm (algorithm* **Max-Flow-GT** *in Figure 2.16) constructs a maximum flow for a given flow network in time $O(n^2m)$.*

PROOF. The correctness of the algorithm has been given in Lemma 2.3.4. The running time of the algorithm is dominated by step 5, for which we give a detailed analysis.

We keep two 2-dimensional arrays $f[1..n, 1..n]$ and $cap[1..n, 1..n]$ for the flow value and the capacity for the original flow network G, respectively, so that the flow value between a pair of vertices, and the capacity of an edge in the residual network G_f can be obtained and modified in constant time. Similarly, we keep the arrays $h[1..n]$ and $e[1..n]$ for the height and excess for vertices in G so that the values can be obtained and modified in constant time.

The residual network G_f is represented by an adjacency list L_f such that for each vertex v in G_f, the edges $[v, w]$ with $h(v) = h(w) + 1$ appear in the front of the list $L_f[v]$. Finally, we also keep a list OF for the vertices u in G with $e[u] > 0$.

With these data structures, the condition in step 5 can by checked in constant time (simply check if the list OF is empty), and step 5.1 takes a constant time to pick a vertex v from the list OF. Since the edges $[v, w]$ with $h(v) = h(w) + 1$ appear in the front of the list $L_f[v]$, in constant time, we can check if the operation **Push** applies to an out-going edge from v.

For each **Push**(u, w) operation, the modification of the flow values and the excess values can be done in constant time. Moreover, if $[w, u]$ was not an edge in G_f (this can be checked by comparing the values $f[w, u]$ and $cap[w, u]$) then the operation **Push**(u, w) creates a new edge $[w, u]$ in the residual network G_f. This new edge $[w, u]$ shoud be added to the *end* of the list $L_f[w]$ since $h(w) = h(u) - 1 \neq h(u) + 1$. In conclusion, each **Push** operation takes time $O(1)$. By Lemmma 2.3.7, the total number of **Push** operations executed by the algorithm **Max-Flow-GT** is bounded by $O(n^2 m)$. Thus, the total time the algorithm **Max-Flow-GT** spends on the **Push** operations is bounded by $O(n^2 m)$.

Now consider the **Lift** operation. A **Lift**(v) operation needs to find a vertex w_0 with minimum $h(w_0)$ in the list $L_f[v]$, which takes time $O(n)$. After increasing the value of $h(v)$, we need to check each in-coming edge $[u, v]$ to v to see if now $h(u) = h(v) + 1$, and to check each out-going edge $[v, w]$ from v to see if now $h(v) = h(w) + 1$. If so, the edge should be moved to the front of the list for the proper vertex in L_f. In any case, all these can be done in time $O(m)$. According to Lemma 2.3.6, the total number of **Lift** operations executed by the algorithm **Max-Flow-GT** is bounded by $2n^2 - 8$. Thus, the total time the algorithm **Max-Flow-GT** spends on the **Lift** operations is also bounded by $O(n^2 m)$.

This concludes that the running time of the algorithm **Max-Flow-GT** is bounded by $O(n^2 m)$. $\square$

We point out that it has been left totally open for the order of the vertices selected by step 5.1 in the algorithm **Max-Flow-GT**, which gives us further opportunity for improving the running time of the algorithm. In fact, with a more careful selection of the vertex v in step 5.1 and a more efficient data structure, it can be shown that running time of the algorithm **Max-Flow-GT** can be improved to $O(nm\log(n^2/m))$. For dense flow networks with $m = \Omega(n^2)$ edges, the bound $O(nm\log(n^2/m))$ is as good as $O(n^3)$, the bound for Karzanov's maximum flow algorithm, while for sparse flow networks with $m = O(n)$ edges, the bound $O(nm\log(n^2/m))$ becomes $O(n^2\log n)$, much lower than $O(n^3)$. A description of this improvement can be found in [55].

2.4 Final remarks

Before closing the chapter, we give several remarks on the Maximum FLow problem. First, we show that the Maximum Flow problem is closely related to a graph cut problem. This is given by the classical *Max-Flow-Min-Cut* theorem. Then, we briefly mention the updated status in the research in maximum flow algorithms.

Max-Flow-Min-Cut theorem

Let $G = (V, E)$ be a directed and positively weighted graph. A *cut* in G is a partition of the vertex set V of G into an orderd pair (V_1, V_2) of two non-empty subsets V_1 and V_2, i.e., $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$. The *weight* of the cut (V_1, V_2) is the sum of the weights of the edges $[u, w]$ with $u \in V_1$ and $w \in V_2$.

The Min-Cut problem is to find a cut of minimum weight for a given directed and positively weighted graph. Formally,

$$\text{Min-Cut} = \langle I_Q, S_Q, f_Q, opt_Q \rangle$$

I_Q: the set of directed and positively weighted graphs G

S_Q: $S_Q(G)$ is the set of cuts for G

f_Q: $f_Q(G, C)$ is equal to the weight of the cut C for G

opt_Q: min

A more restricted Min-Cut problem is on flow networks. We say that (V_1, V_2) is a *cut* for a flow network G if (V_1, V_2) is a cut for G when G is regarded as a directed and positively weighted graph, where the edge weight equals the edge capacity in G, such that the source s of G is in V_1 and the sink t of G is in V_2. We now consider the MIN-CUT problem on flow networks.

Definition 2.4.1 The *capacity* of a cut (V_1, V_2) for a flow network G is the weight of the cut:

$$cap(V_1, V_2) = \sum_{u \in V_1, w \in V_2} cap(v, w)$$

A cut for G is *minimum* if its capacity is the minimum over all cuts for G.

Now we are ready to prove the following classical theorem.

Theorem 2.4.1 (Max-Flow Min-Cut Theorem) *For any flow network $G = (V, E)$, the value of a maximum flow in G is equal to the capacity of a minimum cut for G.*

PROOF. Let f be a flow in the flow network G and let (V_1, V_2) be a cut for G. By the definition, we have $|f| = \sum_{w \in V} f(s, w)$. By the flow conservation property, we also have $\sum_{w \in V} f(v, w) = 0$ for all vertices $v \in V_1 - \{s\}$. Therefore, we have

$$\begin{aligned} |f| &= \sum_{w \in V} f(s, w) = \sum_{v \in V_1, w \in V} f(v, w) \\ &= \sum_{v \in V_1, w \in V_1} f(v, w) + \sum_{v \in V_1, w \in V_2} f(v, w) \end{aligned}$$

By the skew symmetry property, $f(v, w) = -f(w, v)$ for all vertices v and w in V_1. Thus, the first term in the last expression of the above equation is equal to 0. Thus,

$$|f| = \sum_{v \in V_1, w \in V_2} f(v, w) \tag{2.3}$$

By the capacity constraint property of a flow, we have

$$|f| \leq \sum_{v \in V_1, w \in V_2} cap(v, w) = cap(V_1, V_2)$$

Since this inequality holds for all flows f and all cuts (V_1, V_2), we conclude

$$\max\{|f| : f \text{ is a flow in G}\} \leq \min\{cap(V_1, V_2) : (V_1, V_2) \text{ is a cut for G}\}$$

To prove the other direction, let f be a maximum flow in the network G and let G_f be the residual network. By Theorem 2.1.4, there is no path from the source s to the sink t in the residual network G_f. Define V_1 to be the set of vertices that are reachable from the source s in the graph G_f and let V_2 be the set of the rest vertices. Then, $s \in V_1$ and $t \in V_2$ so (V_1, V_2) is a cut for the flow network G. We have

$$cap(V_1, V_2) = \sum_{v \in V_1, w \in V_2} cap(v, w) = \sum_{[v,w] \in E, v \in V_1, w \in V_2} cap(v, w)$$

Consider an edge $[v, w]$ in G such that $v \in V_1$ and $w \in V_2$. Since $[v, w]$ is not an edge in the residual network G_f (otherwise, the vertex w would be reachable from s in G_f), we must have $f(v, w) = cap(v, w)$. On the other hand, for $v \in V_1$ and $w \in V_2$ suppose (v, w) is not an edge in G, then we have $cap(v, w) = 0$ thus $f(v, w) \leq 0$. However, $f(v, w) < 0$ cannot hold since otherwise we would have $cap(v, w) > f(v, w)$ thus $[v, w]$ would have been an edge in the residual graph G_f, contradicting the assumption $v \in V_1$ and $w \in V_2$. Thus, in case (v, w) is not an edge in G and $v \in V_1$ and $w \in V_2$, we must have $f(v, w) = 0$. Combining these discussions, we have

$$ap(V_1, V_2) = \sum_{[v,w] \in E, v \in V_1, w \in V_2} cap(v, w) = \sum_{v \in V_1, w \in V_2} f(v, w)$$

By equation 2.3, the last expression is equal to $|f|$. This proves

$$\max\{|f| : f \text{ is a flow in G}\} \geq \min\{cap(V_1, V_2) : (V_1, V_2) \text{ is a cut for G}\}$$

The proof of the theorem is thus completed. $\square$

The Max-Flow-Min-Cut theorem used to be used to show that a maximum flow in a flow network can be found in finite number of steps (since there are only $2^n - 2$ cuts for a flow network of n vertices). With more efficient maximum flow algorithms being developed, now the theorem can be used in the opposite direction — to find a minimum cut for a flow network. In fact, the proof of Theorem 2.4.1 has described such an algorithm: given a flow network $G = (V, E)$, construct a maximum flow f in G, let V_1 be the set of vertices reachable from the source s in the residual network G_f and let $V_2 = V - V_1$, then (V_1, V_2) is a minimum cut for the flow network G. Thus, the Min-Cut problem on flow networks can be solved in time $O(n^3)$, if we use, for example, Karzanov's maximum flow algorithm to solve the Maximum Flow problem.

Date	Authors	Time Complexity
1960	Edmonds and Karp	$O(nm^2)$
1970	Dinic	$O(n^2m)$
1974	Karzanov	$O(n^3)$
1977	Cherkasky	$O(n^2\sqrt{m})$
1978	Galil	$O(n^{5/3}m^{2/3})$
1980	Sleator and Tarjan	$O(nm\log n)$
1986	Goldberg and Tarjan	$O(nm\log(n^2/m))$

Figure 2.17: Maximum flow algorithms

The MIN-CUT problem for general graphs can be solved via the algorithm for the problem for flow networks: given a general directed and positively weighted graph G, we fix a vertex v_1 in G. Now for each other vertex w, we construct a flow network G_w that is G with v_1 the source and w the sink, and construct another flow network G_w^r that is G with w the source and v_1 the sink. Then we find the minimum cuts for the flow networks G_w and G_w^r. Since in a minimum cut (V_1, V_2) for G as a general graph, the vertex v_1 must be in one of V_1 and V_2 and some vertex w must be in the other, when we construct the minimum cuts for the flow networks G_w and G_w^r for this particular w, we will find a minimum cut for G as a general graph. Thus, the MIN-CUT problem for general graphs can be solved in polynomial time.

We remark that the maximum version of the cut problem, i.e., the MAX-CUT problem that finds a cut of maximum weight in a directed and positively weighted graph, is much more difficult that the MIN-CUT problem. The MAX-CUT problem will be discussed in more detail in later chapters.

Updated status

Since the introduction of the MAXIMUM FLOW problem by Ford and Fulkerson about thirty years ago, efficient solution of the MAXIMUM FLOW problem has been a very active research area in theoretical computer science. In this chapter, we have discussed two major techniques for maximum flow algorithms. Further extension and improvement on these techniques are studied. The table in Figure 2.17 gives a selected sequence of maximum flow algorithms developed over the past thirty years. For further discussion on maximum flow algorithms, the reader is referred to the survey by Goldberg, Tardos, and Tarjan [54].