unblocked vertex v, we examine the edges out of v, beginning with the last edge previously examined, and increase the flow on each edge to which the increasing step applies, until $\Delta f(v) = 0$ or we run out of edges (the balancing fails). Balancing a blocked vertex is similar. With such an implementation the method takes $O(n^2)$ time to find a blocking flow, including the time to topologically order the vertices. Theorem 8.4 gives the $O(n^3)$ time bound for finding a maximum flow. □

When using the wave algorithm to find a maximum flow, we can use the layered structure of the level graphs to find each blocking flow in $O(m + k)$ time, where k is the number of balancings, eliminating the $O(n^2)$ overhead for scanning balanced vertices [25]. This may give an improvement in practice, though the time bound is still $O(n^2)$ in the worst case.

Malhotra, Kumar and Maheshwari [17] suggested another $O(n^2)$-time blocking flow method that is conceptually very simple. Initially we delete from G every vertex and edge not on a path from s to t. We maintain for each vertex v the *potential throughput* of v, defined by

$$thruput\ (v) = \min\left\{\sum_{[u,v]\in E} (cap\ (u, v) - f(u, v)), \sum_{[v,w]\in E} (cap\ (v,w) - f(v, w))\right\}.$$

(To define *thruput* (s) and *thruput* (t), we assume the existence of a dummy edge of infinite capacity from t to s.) To find a blocking flow we repeat the following step until t is not reachable from s:

SATURATING STEP. Let v be a vertex of minimum potential throughput. Send *thruput* (v) units of flow forward from v to t by scanning the vertices in topological order and backward from v to s by scanning the vertices in reverse topological order. Update all throughputs, delete all newly saturated edges from G (this includes either all edges entering or all edges leaving v) and delete all vertices and edges not on a path from s to t.

Although this method is simple, it has two drawbacks. When actually implemented, it is at least as complicated as the wave method. Furthermore, it preferentially sends flow through narrow bottlenecks, which may cause it to perform many more augmentations than necessary. For these reasons we prefer the wave method.

A fourth way to find a blocking flow is to saturate one edge at a time as in Dinic's method, but to reduce the time per edge saturation by using an appropriate data structure to keep track of the flow. Galil and Naamad [11] and Shiloach [21] discovered a method of this kind that runs in $O(m(\log n)^2)$ time. Sleator and Tarjan [23], [24] improved the bound to $O(m \log n)$, inventing the data structure of Chapter 5 for this purpose. We conclude this section by describing their method.

Recall that the data structure of Chapter 5 allows us to represent a collection of vertex-disjoint rooted trees, each of whose vertices has a real-valued cost, under the following operations, each of which takes $O(\log n)$ amortized time:

maketree (v): Create a new tree containing the single vertex v, of cost zero.
findroot (v): Return the root of the tree containing vertex v.

findcost (v): Return the pair $[w, x]$, where x is the minimum cost of a vertex on the tree path from v to findroot (v) and w is the last vertex on this path of cost x.

addcost (v, x): Add x to the cost of every vertex on the tree path from v to findroot (v).

link (v, w): Combine the two trees containing vertices v and w by adding the edge $[v, w]$ (v must be a root).

cut (v): Divide the tree containing vertex v into two trees by deleting the edge out of v (v must be a nonroot).

To find a blocking flow, we maintain for each vertex v a current edge $[v, p(v)]$ on which it may be possible to increase the flow. These edges define a collection of trees. (Some vertices may not have a current edge.) The cost of a vertex v is $cap\,(v, p(v)) - f(v, p(v))$ if v is not a tree root, *huge* if v is a tree root, where *huge* is a constant chosen larger than the sum of all the edge capacities. The following steps are a reformulation of Dinic's algorithm using the five tree operations. We find a blocking flow by first executing maketree (v) followed by addcost $(v, huge)$ for all vertices, then going to *advance* and proceeding as directed.

Advance. Let v = findroot (s). If there is no edge out of v, go to *retreat.* Otherwise, let $[v, w]$ be an edge out of v. Perform addcost $(v, cap\,(v, w) - huge)$ followed by link (v, w). Define $p(v)$ to be w. If $w \neq t$, repeat *advance;* if $w = t$, go to *augment.*

Augment. Let $[v, \Delta]$ = findcost (s). Perform addcost $(s, -\Delta)$. Go to *delete.*

Delete. Perform cut (v) followed by addcost $(v, huge)$. Define $f(v, p(v)) = cap\,(v, p(v))$. Delete $[v, p(v)]$ from the graph. Let $[v, \Delta]$ = findcost (s). If $\Delta = 0$, repeat *delete;* otherwise go to *advance.*

Retreat. If $v = s$ halt. Otherwise, for every edge $[u, v]$, delete $[u, v]$ from the graph and, if $p(u) \neq v$, define $f(u, v) = 0$; if $p(u) = v$, perform cut (u), let $[u, \Delta]$ = findcost (u), perform addcost $(u, huge - \Delta)$, and define $f(u, v) = cap\,(u, v) - \Delta$. After deleting all edges $[u, v]$, go to *advance.*

Once the algorithm halts, we use cut, findcost, and addcost as in *retreat* to find the flow on every remaining edge.

THEOREM 8.10. *The Sleator–Tarjan algorithm correctly finds a blocking flow in $O(m \log n)$ time and a maximum flow in $O(nm \log n)$ time.*

Proof. The correctness of the method is immediate. There are $O(m)$ tree operations, giving a time bound of $O(m \log n)$ to find a blocking flow. The time bound for a maximum flow follows from Theorem 8.4. □

It is intriguing to contemplate the possibility of implementing the wave method using the data structure for cutting and linking trees, thereby obtaining an algorithm as fast as any known method on both sparse and dense graphs. We leave this as an open problem; we conjecture that a time bound of $O(m \log (n^2/m))$ for finding a blocking flow can be obtained in this way.

8.4. Minimum cost flows. The idea of augmenting paths extends to a more general network flow problem. Let G be a network such that each edge $[v, w]$ has a