the blocking flow method, the Goldfarb-Hao algorithm does not necessarily augment along shortest augmenting paths. In their analysis, Goldfarb and Hao use a variant of distance labeling and a variant of the *relabeling* operation mentioned above. Dynamic trees can be used to obtain an $O(nm \log n)$-time implementation of their algorithm [42].

The best currently known sequential bounds for the maximal flow problem are $O(nm \log(n^2/m))$ and $O(nm \log(\frac{n}{m}\sqrt{\log U} + 2))$. Note that, although the running times of the known algorithms come very close to an $O(mn)$ bound, the existence of a maximum flow algorithm that meets this bound remains open.

With the increasing interest in parallel computing, parallel and distributed algorithms for the maximum flow problem have received a great deal of attention. The first parallel algorithm for the problem, due to Shiloach and Vishkin [93], runs in $O(n^2 \log n)$ time on an n-processor PRAM [28] and uses $O(n)$ memory per processor. In a synchronous distributed model of computation, this algorithm runs in $O(n^2)$ time using $O(n^3)$ messages and $O(n)$ memory per processor. The algorithm of Goldberg [40, 41, 46] uses less memory than that of Shiloach and Vishkin: $O(m)$ memory for the PRAM implementation and $O(\Delta)$ memory per processor for the distributed implementation (where Δ is the processor degree in the network). The time, processor, and message bounds of this algorithm are the same as those of the Shiloach-Vishkin algorithm. Ahuja and Orlin [3] developed a PRAM implementation of their excess-scaling algorithm. The resource bounds are $\lceil m/n \rceil$ processors, $O(m)$ memory, and $O(n^2 \log n \log U)$ time. Cheriyan and Maheshwari [16] proposed a synchronous distributed implementation of the largest-label algorithm that runs in $O(n^2)$ time using $O(\Delta)$ memory per processor and $O(n^2\sqrt{m})$ messages.

For a long time, the primal network simplex method was the method of choice in practice. A study of Goldfarb and Grigoriadis [51] suggested that the algorithm of Dinic [21] performs better than the network simplex method and better than the later algorithms based on the blocking flow method. Recent studies of Ahuja and Orlin (personal communication) and Grigoriadis (personal communication) show superiority of various versions of the push-relabel method to Dinic's algorithm. An experimental study of Goldberg [41] shows that a substantial speedup can be achieved by implementing the FIFO algorithm on a highly parallel computer.

More efficient algorithms have been developed for the special case of planar networks. Ford and Fulkerson [27] have observed that the the maximum flow problem on a planar network is related to a shortest path problem on the planar dual of the network. The algorithms in [8, 27, 30, 56, 57, 60, 64, 76, 88] make clever use of this observation.

2.2 A Generic Algorithm

In this section we describe the generic push-relabel algorithm [41, 45, 46]. First, however, we need the following definition. For a given preflow f, a *distance labeling* is a function d from the vertices to the non-negative integers such that

$d(t) = 0$, $d(s) = n$, and $d(v) \leq d(w) + 1$ for all residual arcs (v, w). The intuition behind this definition is as follows. Define a *distance graph* G_f^* as follows. Add an arc (s, t) to G_f. Define the length of all residual arcs to be equal to one and the length of the arc (s, t) to be n. Then d is a "locally consistent" estimate on the distance to the sink in the distance graph. (In fact, it is easy to show that d is a lower bound on the distance to the sink.) We denote by $d_{G_f^*}(v, w)$ the distance from vertex v to vertex w in the distance graph. The generic algorithm maintains a preflow f and a distance labeling d for f, and updates f and d using *push* and *relabel* operations. To describe these operations, we need the following definitions. We say that a vertex v is *active* if $v \notin \{s, t\}$ and $e_f(v) > 0$. Note that a preflow f is a flow if and only if there are no active vertices. An arc (v, w) is *admissible* if $(v, w) \in E_f$ and $d(v) = d(w) + 1$.

The algorithm begins with the preflow f that is equal to the arc capacity on each arc leaving the source and zero on all arcs not incident to the source, and with some initial labeling d. The algorithm then repetitively performs, in any order, the *update operations*, *push* and *relabel*, described in Figure 2.2. When there are no active vertices, the algorithm terminates. A summary of the algorithm appears in Figure 2.1.

```
procedure generic (V, E, u);
    [initialization]
    ∀(v, w) ∈ E do begin
        f(v, w) ← 0;
        if v = s then f(s, w) ← u(s, w);
        if w = s then f(v, s) ← −u(s, v);
    end;
    ∀w ∈ V do begin
        e_f(w) ← Σ_{(v,w)∈E} f(v, w);
        if w = s then d(w) = n else d(w) = 0;
    end;
    [loop]
    while ∃ an active vertex do
        select an update operation and apply it;
    return(f);
end.
```

Fig. 2.1 The generic maximum flow algorithm

The update operations modify the preflow f and the labeling d. A *push* from v to w increases $f(v, w)$ and $e_f(w)$ by up to $\delta = \min\{e_f(v), u_f(v, w)\}$, and decreases $f(w, v)$ and $e_f(v)$ by the same amount. The push is *saturating* if $u_f(v, w) = 0$ after the push and *nonsaturating* otherwise. A *relabeling* of v sets the label of v equal to the largest value allowed by the valid labeling constraints.

push(v, w).
Applicability: v is active **and** (v, w) is admissible.
Action: send $\delta \in (0, \min(e_f(v), u_f(v, w))]$ units of flow from v to w.

relabel(v).
Applicability: either s or t is reachable from v in G_f **and** $\forall w \in V\ u_f(v, w) = 0$ or $d(w) \geq d(v)$.
Action: replace $d(v)$ by $\min_{(v,w)\in E_f}\{d(w)\} + 1$.

Fig. 2.2 The update operations. The *pushing* operation updates the preflow, and the *relabeling* operation updates the distance labeling. Except for the excess scaling algorithm, all algorithms discussed in this section push the maximum possible amount δ when doing a push

There is one part of the algorithm we have not yet specified: the choice of an initial labeling d. The simplest choice is $d(s) = n$ and $d(v) = 0$ for $v \in V - \{s\}$. A more accurate choice (indeed, the most accurate possible choice) is $d(v) = d_{G_f}(v, t)$ for $v \in V$, where f is the initial preflow. The latter labeling can be computed in $O(m)$ time using backwards breadth-first searches from the sink and from the source in the residual graph. The resource bounds we shall derive for the algorithm are correct for any valid initial labeling. To simplify the proofs, we assume that the algorithm starts with the simple labeling. In practice, it is preferable to start with the most accurate values of the distance labels, and to update the distance labels periodically by using backward breadth-first search.

Remark. By giving priority to relabeling operations, it is possible to maintain the following invariant: Just before a push, d gives the exact distance to t in the distance graph. Furthermore, it is possible to implement the relabeling operations so that the total work done to maintain the distance labels is $O(nm)$ (see e.g. [46]). Since the running time bounds derived in this section are $\Omega(nm)$, one can assume that the relabeling is done in this way. In practice, however, maintaining exact distances is expensive; a better solution is to maintain a valid distance labeling and periodically update it to the exact labeling.

Next we turn our attention to the correctness and termination of the algorithm. Our proof of correctness is based on Theorem 1.2.1. The following lemma is important in the analysis of the algorithm.

2.2.1 Lemma. *If f is a preflow and v is a vertex with positive excess, then the source s is reachable from v in the residual graph G_f.*

Using this lemma and induction on the number of update operations, it can be shown that one of the two update operations must be applicable to an active vertex, and that the operations maintain a valid distance labeling and preflow.

2.2.2 Theorem. *Suppose that the algorithm terminates. Then the preflow f is a maximum flow.*

Proof. When the algorithm terminates, all vertices in $V - \{s, t\}$ must have zero excess, because there are no active vertices. Therefore f must be a flow. We show that if f is a preflow and d is a valid labeling for f, then the sink t is not reachable from the source s in the residual graph G_f. Then Theorem 1.2.1 implies that the algorithm terminates with a maximum flow.

Assume by way of contradiction that there is an augmenting path $s = v_0, v_1, \ldots, v_l = t$. Then $l < n$ and $(v_i, v_{i+1}) \in E_f$ for $0 \leq i < l$. Since d is a valid labeling, we have $d(v_i) \leq d(v_{i+1}) + 1$ for $0 \leq i < l$. Therefore, we have $d(s) \leq d(t) + l < n$, since $d(t) = 0$, which contradicts $d(s) = n$. □

The key to the running time analysis of the algorithm is the following lemma, which shows that distance labels cannot increase too much.

2.2.3 Lemma. *At any time during the execution of the algorithm, for any vertex $v \in V$, $d(v) \leq 2n - 1$.*

Proof. The lemma is trivial for $v = s$ and $v = t$. Suppose $v \in V - \{s, t\}$. Since the algorithm changes vertex labels only by means of the *relabeling* operation, it is enough to prove the lemma for a vertex v such that s or t is reachable from v in G_f. Thus there is a simple path from v to s or t in G_f. Let $v = v_0, v_1, \ldots, v_l$ be such a path. The length l of the path is at most $n - 1$. Since d is a valid labeling and $(v_i, v_{i+1}) \in E_f$, we have $d(v_i) \leq d(v_{i+1}) + 1$. Therefore, since $d(v_l)$ is either n or 0, we have $d(v) = d(v_0) \leq d(v_l) + l \leq n + (n - 1) = 2n - 1$. □

Lemma 2.2.3 limits the number of relabeling operations, and allows us to amortize the work done by the algorithm over increases in vertex labels. The next two lemmas bound the number of relabelings and the number of saturating pushes.

2.2.4 Lemma. *The number of relabeling operations is at most $2n - 1$ per vertex and at most $(2n - 1)(n - 2) < 2n^2$ overall.*

2.2.5 Lemma. *The number of saturating pushes is at most nm.*

Proof. For an arc $(v, w) \in E$, consider the saturating pushes from v to w. After one such push, $u_f(v, w) = 0$, and another such push cannot occur until $d(w)$ increases by at least 2, a push from w to v occurs, and $d(v)$ increases by at least 2. If we charge each saturating push from v to w except the first to the preceding label increase of v, we obtain an upper bound of n on the number of such pushes. □

The most interesting part of the analysis is obtaining a bound on the number of nonsaturating pushes. For this we use amortized analysis and in particular the *potential function* technique (see e.g. [98]).

2.2.6 Lemma. *The number of nonsaturating pushing operations is at most $2n^2m$.*

Proof. We define the *potential* Φ of the current preflow f and labeling d by the formula $\Phi = \sum_{\{v|v \text{ is active}\}} d(v)$. We have $0 \leq \Phi \leq 2n^2$ by Lemma 2.2.3. Each

nonsaturating push, say from a vertex v to a vertex w, decreases Φ by at least one, since $d(w) = d(v) - 1$ and the push makes v inactive. It follows that the total number of nonsaturating pushes over the entire algorithm is at most the sum of the increases in Φ during the course of the algorithm, since $\Phi = 0$ both at the beginning and at the end of the computation. Increasing the label of a vertex v by an amount k increases Φ by k. The total of such increases over the algorithm is at most $2n^2$. A saturating push can increase Φ by at most $2n - 2$. The total of such increases over the entire algorithm is at most $(2n-2)nm$. Summing gives a bound of at most $2n^2 + (2n-2)nm \leq 2n^2m$ on the number of nonsaturating pushes. □

2.2.7 Theorem [46]. *The generic algorithm terminates after $O(n^2m)$ update operations.*

Proof. Immediate from Lemmas 2.2.4, 2.2.5, and 2.2.6. □

The running time of the generic algorithm depends upon the order in which update operations are applied and on implementation details. In the next sections we explore these issues. First we give a simple implementation of the generic algorithm in which the time required for the nonsaturating pushes dominates the overall running time. Sections 2.3 and 2.4 specify orders of the update operations that decrease the number of nonsaturating pushes and permit $O(n^3)$ and $O(mn + n^2 \log U)$-time implementations. Section 2.5 explores an orthogonal approach. It shows how to use sophisticated data structures to reduce the time per nonsaturating push rather than the number of such pushes.

2.3 Efficient Implementation

Our first step toward an efficient implementation is a way of combining the update operations locally. We need some data structures to represent the network and the preflow. We call an unordered pair $\{v, w\}$ such that $(v, w) \in E$ an *edge* of G. We associate the three values $u(v, w)$, $u(w, v)$, and $f(v, w)(= -f(w, v))$ with each edge $\{v, w\}$. Each vertex v has a list of the incident edges $\{v, w\}$, in fixed but arbitrary order. Thus each edge $\{v, w\}$ appears in exactly two lists, the one for v and the one for w. Each vertex v has a *current edge* $\{v, w\}$, which is the current candidate for a pushing operation from v. Initially, the current edge of v is the first edge on the edge list of v. The main loop of the implementation consists of repeating the *discharge* operation described in Figure 2.3 until there are no active vertices. (We shall discuss the maintenance of active vertices later.) The *discharge* operation is applicable to an active vertex v. This operation iteratively attempts to push the excess at v through the current edge $\{v, w\}$ of v if a pushing operation is applicable to this edge. If not, the operation replaces $\{v, w\}$ as the current edge of v by the next edge on the edge list of v; or, if $\{v, w\}$ is the last edge on this list, it makes the first edge on the list the current one and relabels v. The operation stops when the excess at v is reduced to zero or v is relabeled.


```
discharge(v).
Applicability: v is active.
Action:        let {v, w} be the current edge of v;
               time-to-relabel ← false;
               repeat
                  if (v, w) is admissible then push(v, w)
                  else
                     if {v, w} is not the last edge on the edge list of v then
                        replace {v, w} as the current edge of v by the
                        next edge on the list
                     else begin
                        make the first edge on the edge list of v the current edge;
                        time-to-relabel ← true;
                     end;
               until e_f(v) = 0 or time-to-relabel;
               if time-to-relabel then relabel(v);
```

Fig. 2.3 The discharge operation

The following lemma shows that *discharge* does relabeling correctly; the proof of the lemma is straightforward.

2.3.1 Lemma. *The discharge operation does a relabeling only when the relabeling operation is applicable.*

2.3.2 Lemma. *The version of the generic push/relabel algorithm based on discharging runs in $O(nm)$ time plus the total time needed to do the nonsaturating pushes and to maintain the set of active vertices.*

Any representation of the set of active vertices that allows insertion, deletion, and access to some active vertex in $O(1)$ time results in an $O(n^2m)$ running time for the discharge-based algorithm, by Lemmas 2.2.6 and 2.3.2. (Pushes can be implemented in $O(1)$ time per push.)

By processing active vertices in a more restricted order, we obtain improved performance. Two natural orders were suggested in [45, 46]. One, the *FIFO algorithm*, is to maintain the set of active vertices as a queue, always selecting for discharging the front vertex on the queue and adding newly active vertices to the rear of the queue. The other, the *largest-label algorithm*, is to always select for discharging a vertex with the largest label. The FIFO algorithm runs in $O(n^3)$ time [45, 46] and the largest-label algorithm runs in $O(n^2\sqrt{m})$ time [16]. We shall derive an $O(n^3)$ time bound for both algorithms, after first describing in a little more detail how to implement largest-label selection.

The implementation maintains an array of sets B_i, $0 \leq i \leq 2n - 1$, and an index b into the array. Set B_i consists of all active vertices with label i, represented as a doubly-linked list, so that insertion and deletion take $O(1)$ time. The index b is the largest label of an active vertex. During the initialization, when the arcs

```
procedure process-vertex;
    remove a vertex v from B_b;
    old-label ← d(v);
    discharge(v);
    add each vertex w made active by the discharge to B_d(w);
    if d(v) ≠ old-label then begin
        b ← d(v);
        add v to B_b;
    end
    else if B_b = ∅ then b ← b − 1;
end.
```

Fig. 2.4 The *process-vertex* procedure

going out of the source are saturated, the resulting active vertices are placed in B_0, and b is set to 0. At each iteration, the algorithm removes a vertex from B_b, processes it using the *discharge* operation, and updates b. The algorithm terminates when b becomes negative, *i.e.*, when there are no active vertices. This processing of vertices, which implements the *while* loop of the generic algorithm, is described in Figure 2.4.

To understand why the *process-vertex* procedure correctly maintains b, note that *discharge*(v) either relabels v or gets rid of all excess at v, but not both. In the former case, v is the active vertex with the largest distance label, so b must be increased to $d(v)$. In the latter case, the excess at v has been moved to vertices with distance labels of $b-1$, so if B_b is empty, then b must be decreased by one. The total time spent updating b during the course of the algorithm is $O(n^2)$.

The bottleneck in both the FIFO method and the largest-label method is