SIAM J. COMPUT. © 1996 Society for Industrial and Applied Mathematics
Vol. 25, No. 6, pp. 1144–1170, December 1996 002

AN $o(n^3)$-TIME MAXIMUM-FLOW ALGORITHM[*]

JOSEPH CHERIYAN[†], TORBEN HAGERUP[‡], AND KURT MEHLHORN[‡]

Abstract. We show that a maximum flow in a network with n vertices can be computed deterministically in $O(n^3/\log n)$ time on a uniform-cost RAM. For dense graphs, this improves the previous best bound of $O(n^3)$.

The bottleneck in our algorithm is a combinatorial problem on (unweighted) graphs. The number of operations executed on flow variables is $O(n^{8/3}(\log n)^{4/3})$, in contrast with $\Omega(nm)$ flow operations for all previous algorithms, where m denotes the number of edges in the network. A randomized version of our algorithm executes $O(n^{3/2}m^{1/2}\log n + n^2(\log n)^2/\log(2+n(\log n)^2/m))$ flow operations with high probability.

For the special case in which all capacities are integers bounded by U, we show that a maximum flow can be computed deterministically using $O(n^{3/2}m^{1/2} + n^2(\log U)^{1/2} + \log U)$ flow operations and $O(\min\{nm, n^3/\log n\} + n^2(\log U)^{1/2} + \log U)$ time. We finally argue that several of our results yield parallel algorithms with optimal speedup.

Key words. network flow, maximum flow, graph algorithm, scaling, preflow-push algorithm, current-edge problem, dynamic tree

AMS subject classifications. 68P05, 68Q20, 68Q22, 68Q25, 68R05, 90B10, 90C35

1. Introduction.

The fastest algorithm predating this paper for computing a maximum flow in a network with n vertices and m edges, even allowing randomization, has an expected running time of $O(\min\{nm\log n, nm + n^2(\log n)^2\})$ [CH95]. Despite intensive research for over three decades, no algorithm with a running time of $o(nm)$ has ever been reported for any combination of n and m. This is true even for networks with integer capacities, provided that the maximum capacity U is moderately large, say $U = \Omega(n)$ [AOT89].

Our main result is a maximum-flow algorithm that runs in $O(n^3/\log n)$ time. For dense networks with $m = \omega(n^2/\log n)$ this is $o(nm)$. We also slightly improve the best previous results for sparse networks with $m = o(n(\log n)^2)$ and $m = \omega(n\log n/\log\log n)$ and match the best previous results for other ranges of m through simpler algorithms. Our algorithms are *strongly polynomial*; this means, roughly speaking, that the running time is bounded by a polynomial in the number of vertices in the network independently of the capacities of the edges (see [GLS88] for a more careful definition). Table 1 below summarizes the running times of our algorithms for different combinations of n and m.

Subsequently to our work, King, Rao, and Tarjan [KRT94] extended the range of network densities for which the performance of randomized maximum-flow algorithms can be matched by deterministic algorithms. Their deterministic algorithm runs in $O(nm\log_{m/(n\log n)} n)$ time on networks with n vertices and $m > n\log n$ edges. This running time is $O(nm)$ when $m \ge n^{1+\epsilon}$ for some fixed $\epsilon > 0$, while it is strictly larger than our running time for all other network densities.

[*]Received by the editors November 20, 1991; accepted for publication (in revised form) February 8, 1995. This research was partially supported by EC ESPRIT II Basic Research Actions Program contract 3075 (project ALCOM). A preliminary version of this paper was presented at the 17th International Colloquium on Automata, Languages and Programming (ICALP) in July 1990.

[†]Department of Combinatorics and Optimization, University of Waterloo, Waterloo, ON N2L 3G1, Canada.

[‡]Max-Planck-Institut für Informatik, D-66123 Saarbrücken, Germany (torben@mpi-sb.mpg.de).

TABLE 1

Strongly polynomial maximum-flow algorithms for different combinations of n, the number of vertices, and m, the number of edges. The first column gives the order of the bound on the running time of each algorithm, the second column indicates the range of network densities for which the algorithm is superior to the other algorithms, the third column states whether the algorithm is deterministic or randomized, and the last column gives the source of the algorithm and a corresponding theorem in the present paper. The new results appear in lines 2 and 5 of the body of the table.

Running Time	Range	Model	Source
$nm \log n$	$m \le n\frac{\log n}{\log\log n}$	deterministic	[ST83] and this paper, Theorem 8.1(a)
$\frac{n^2(\log n)^2}{\log(2+n(\log n)^2/m)}$	$n\frac{\log n}{\log\log n} \le m \le n(\log n)^2$	randomized	this paper, Theorem 8.1(c)
nm	$n(\log n)^2 \le m \le n^{5/3}\log n$	randomized	[CH95] and this paper, Theorem 8.1(c)
nm	$n^{5/3}\log n \le m \le \frac{n^2}{\log n}$	deterministic	[Al90] and this paper, Theorem 8.1(b)
$\frac{n^3}{\log n}$	$\frac{n^2}{\log n} \le m \le n^2$	deterministic	this paper, Theorem 8.2

Our algorithm is based on earlier work in [CH89], [GT88], and [AO89], all of which in turn use the generic maximum-flow algorithm of Goldberg and Tarjan [GT88], which works by manipulating a so-called *preflow* [Ka74] in the given network. We design an extension of the generic algorithm, called the *incremental generic algorithm*, which uses a new operation called *add edge*. The new algorithm manipulates a preflow in a subnetwork and, as the execution progresses, gradually adds the remaining edges to the current subnetwork.

Adding the edges in the order of decreasing capacities allows instances of the incremental generic algorithm to save on the number of operations on flow variables. In particular, the number of flow operations executed by our main algorithm is $O(n^{8/3}(\log n)^{4/3})$. To the best of our knowledge, all previous algorithms execute $\Omega(nm)$ flow operations. Using randomization, we can do even better: a maximum flow can be computed using $O(n^{3/2}m^{1/2}\log n + n^2(\log n)^2/\log(2+n(\log n)^2/m))$ flow operations with high probability. In fact, our deterministic algorithm is obtained from the randomized algorithm by applying a derandomization technique due to Alon [Al90]. Our analysis of flow operations is based on a novel potential argument.

The bottleneck in our algorithms turns out to be a simple combinatorial problem on a dynamically changing (unweighted) graph, that of repeatedly identifying the so-called *current edge* of a given vertex. Indeed, given a sufficiently efficient solution to the current-edge problem, the running time of each of our algorithms would match the number of flow operations. A straightforward solution to the current-edge problem contributes $\Theta(nm)$ time to the running time of the maximum-flow algorithms. The idea behind our improvement of this bound for dense networks, by a factor of $\Theta(\log n)$, is to represent the graph by its adjacency matrix and to partition the matrix into $1 \times \lfloor \log n \rfloor$ submatrices. A submatrix can be processed in constant time by table look-up during the search for a current edge.

Our ideas also apply to networks with integer capacities. For networks with integer capacities bounded by U, one of the fastest algorithms known is the wave scaling algorithm of [AOT89], which runs in time $O(nm + n^2(\log U)^{1/2} + \log U)$.

This algorithm refines the excess scaling algorithm of [AO89], whose running time is $O(nm + n^2 \log U)$. For neither algorithm is a better bound than the running time known for the number of flow operations. We give incremental versions of both algorithms and show how to replace the term nm by $\min\{nm, n^3/\log n\}$ in the bound for the running time and by $n^{3/2}m^{1/2}$ in the bound for the number of flow operations.

The paper is organized as follows. Basic definitions are given in §2. The incremental generic algorithm and the current-edge problem are introduced in §3. The incremental excess scaling and wave scaling algorithms for networks with integer capacities are described in §§4 and 5, respectively. Section 6 discusses solutions to the current-edge problem. The strongly polynomial algorithm is presented in §7 and analyzed in §§7 and 8. Section 9 discusses parallel versions of our algorithms, and §10 states a number of open problems. Readers with an exclusive interest in the strongly polynomial algorithm can skip §§4 and 5 almost entirely. The only material in these sections needed later is the definition of γ-fooling height and its properties (F1)–(F5), given after the proof of Lemma 4.1.

2. Definitions and notation.

For every set V and every $e = (v, w) \in V \times V$, let *tail*$(e) = v$, *head*$(e) = w$, and *rev*$(e) = (w, v)$. v and w are the *tail* of e and the *head* of e, respectively. A *network* is a tuple $G = (V, E, cap, s, t)$, where (V, E, cap) is an edge-weighted directed graph, *cap* maps each edge in E to a nonnegative real number called its *capacity*, and s and t are distinct vertices in V called the *source* and the *sink*, respectively. We assume that E is *symmetric* (i.e., *rev*$(e) \in E$ for all $e \in E$) and without self loops (i.e., $v \neq w$ for all $(v, w) \in E$). In order to make the notation less cumbersome, we omit one pair of brackets from expressions such as "*cap*$((v, w))$."

A *preflow* in G is a function $f : E \to \mathbb{R}$ with the following properties:

(1) $f(rev(e)) = -f(e)$ for all $e \in E$ (antisymmetry constraint),

(2) $f(e) \leq cap(e)$ for all $e \in E$ (capacity constraint),

(3) $\sum_{e \in E: head(e) = v} f(e) \geq 0$ for all $v \in V \backslash \{s\}$ (nonnegativity constraint).

A preflow f in G is a *flow* if $\sum_{e \in E: head(e) = v} f(e) = 0$ for all $v \in V \backslash \{s, t\}$ (flow conservation constraint). The *value* of f is $\sum_{e \in E: head(e) = t} f(e)$, i.e., the net flow into t, and a *maximum flow* in G is a flow in G of maximum value. An edge $e \in E$ is *residual* (with respect to f) if $f(e) < cap(e)$. A *push* on e of *value* $c \in \mathbb{R}$ is an increase in $f(e)$ by c. The push is *saturating* iff $f(e) = cap(e)$ afterwards. A push on an edge (v, w) is also called a push *out of* v and a push *into* w. A *labeling* of G is a function $d : V \to \mathbb{N} \cup \{0\}$. The labeling is *valid* for G and a preflow f in G exactly if $d(v) \leq d(w) + 1$ for every edge $(v, w) \in E$ that is residual with respect to f. Our algorithms operate with the concept of an *undirected edge*, i.e., a pair $\{v, w\}$, where $(v, w) \in E$, which intuitively we identify with the pair $\{(v, w), (w, v)\}$ of two antiparallel directed edges. For all symmetric subsets E' of E, let $\overline{E'} = \{\{v, w\} : (v, w) \in E'\}$ be the corresponding set of undirected edges. A push on an undirected edge $\{v, w\} \in \overline{E}$ is a push on one of the edges (v, w) or (w, v). The *capacity* of an undirected edge $\{v, w\}$ is defined as $cap(\{v, w\}) = cap(v, w) + cap(w, v)$. We assume without loss of generality that $cap(\{v, w\}) > 0$ for all $\{v, w\} \in \overline{E}$.

Our algorithms are formulated in the traditional model for the study of problems on networks. They use two data types for numerical values: *integer* and *flow value*. Capacities and flow values are represented by objects of type *flow value*, on which the only allowed arithmetical operations are addition and subtraction, and all other quantities are represented by objects of type *integer*, on which we allow addition, subtraction, multiplication, and integer division. In addition, we assume for both data types standard operations for comparison, data movement, the constant 1, etc. For n-vertex input networks, we allow integers of absolute value $n^{O(1)}$; i.e., we allow a word size of $O(\log n)$ bits. We charge constant time for each basic operation on either type (uniform cost measure). In keeping with common usage, we employ the term "flow operation" to mean any operation on objects of type *flow value*. In our randomized algorithms we assume in addition that generating a random integer takes constant time. More precisely, we assume that for every given integer k with $1 \le k \le n$, a random integer drawn from the uniform distribution over $\{1,\dots,k\}$ and independent of all other such random integers can be obtained in constant time.

We use "log" to denote the logarithm to base 2, and we assume lists to be implemented as a data type with operations *first* and *pop* (among others). Given a list L, *first*(L) returns the first element of L, and *pop*(L) removes the first element of L and returns it.

3. The incremental generic algorithm.

In this section, we generalize the generic maximum-flow algorithm of [GT88] by extending it to include one additional operation, *add edge*.

The goal of the algorithm is to compute a maximum flow in a network $G = (V, E, cap, s, t)$. Let $n = |V|$ and $m = |E|$. In order to avoid trivialities, we assume that $m \ge n \ge 3$. Let $V^+ = V\backslash\{s,t\}$. The main variables used by the incremental generic algorithm are the following:

(1) A network $G^* = (V, E^*, cap^*, s, t)$, where $E^* \subseteq E$ is symmetric and cap^* is the restriction of cap to E^*. G^* is the *current network*, on which the algorithm operates. $E^* = \emptyset$ initially, and the edges in E are gradually added to E^*.

(2) A preflow $f : E^* \to \mathbb{R}$, which gradually evolves into a maximum flow in G.

(3) A labeling $d : V \to \mathbb{N} \cup \{0\}$, valid for f and G^*.

An edge $(v, w) \in E^*$ is called *eligible* exactly if it is residual with respect to f and $d(v) = d(w) + 1$. For all $e \in E^*$, the *residual capacity* of e is defined as $rescap(e) = cap(e) - f(e)$, and for all $v \in V$, the *excess* of v is defined as $excess(v) = \sum_{e\in E^*:head(e)=v} f(e)$.

We now briefly review the generic maximum-flow algorithm of [GT88], which works on the complete network throughout the execution. The labeling d and the preflow f are initialized as follows: $d(s) = n$, $d(v) = 0$ for all $v \in V\backslash\{s\}$, $f(e) = cap(e)$ for all edges e with tail s, $f(e) = -cap(rev(e))$ for all edges e with head s, and $f(e) = 0$ for all other edges e. Note that the initial labeling is valid for the initial preflow. The algorithm repeatedly picks some vertex $v \in V^+$ with positive excess and performs either a push out of v or a relabeling of v. More precisely, if there is an eligible edge with tail v then a push is performed on one such edge, and if there is no eligible edge with tail v then $d(v)$ is increased by one. This maintains the validity of d, which is crucial for the analysis; cf. Lemmas 3.3 and 3.6 below. The algorithm terminates when there is no vertex in V^+ with positive excess.

In the incremental generic algorithm, we start with $d(s) = n$, $d(v) = 0$ for all $v \in V\backslash\{s\}$, and $E^* = \emptyset$, and we gradually add the edges in E to E^*. This creates a problem, however. The validity of d is endangered whenever an edge (v, w) with $d(v) > d(w) + 1$ is added to E^*. We overcome this difficulty by adopting a more conservative rule than that of [GT88] for sending flow out of a vertex. Namely, define the *visible excess*, $excess^*(v)$, of a vertex $v \in V$ by

$$excess^*(v) = excess(v) - \sum_{e\in E\backslash E^*:tail(e)=v} cap(e)$$

and relabel a vertex or send flow out of it only if its visible excess is positive and remains nonnegative (this use of visible excess is in some ways similar to the use of "available excess" in [AOT89]). We will show below (Lemma 3.1) that this rule guarantees that $excess^*(v) \geq 0$ whenever $d(v) > 0$. In particular, when an edge (v, w) with $d(v) > d(w)$ is added to E^*, it can be saturated immediately using the excess available at v, so that d remains valid. We now give the details.

The algorithm maintains the current network G^*, the preflow f, the labeling d, and the functions $excess(v)$ and $excess^*(v)$. Although the functions *excess* and *excess** in principle can be computed from f and E^*, efficiency dictates that they must be represented explicitly. In the description of the algorithm, however, we omit this trivial elaboration.

Since f is by definition antisymmetric, low-level flow manipulation is carried out by the procedure

```
PROCEDURE setflow(e: edge; c: real);
  $f(e) := c$; $f(rev(e)) := -c$;
```

with the special case

```
PROCEDURE saturate(e: edge);
  setflow(e, cap(e));
```

The main routines of the incremental generic algorithm and the algorithm itself follow.

```
PROCEDURE push(e: edge; c: real);
Precondition: $e = (v, w) \in E^*$, $v \in V^+$, $e$ is eligible, and $0 < c \leq \min\{excess^*(v), rescap(e)\}$.
  setflow(e, $f(e) + c$);

PROCEDURE relabel(v: vertex);
Precondition: $v \in V^+$, $excess^*(v) > 0$, and no edge in $E^*$ with tail $v$ is eligible.
  $d(v) := d(v) + 1$;

PROCEDURE add edge($\{v, w\}$: undirected edge);
Precondition: $\{(v, w), (w, v)\} \subseteq E \backslash E^*$.
  $E^* := E^* \cup \{(v, w), (w, v)\}$;
  if $d(v) > d(w)$ then saturate$(v, w)$ fi;
  if $d(w) > d(v)$ then saturate$(w, v)$ fi;

PROCEDURE generic initialize;
  for all $e \in E$ do setflow$(e, 0)$ od; (* zero flow is default for new edges *)
  for all $v \in V \backslash \{s\}$ do $d(v) := 0$ od; $d(s) := n$;
  $E^* := \emptyset$;

INCREMENTAL GENERIC ALGORITHM:
  generic initialize;
  while $\max\{excess^*(v) : v \in V^+\} > 0$ or $E^* \neq E$
  do
    Execute some push, relabel, or add edge operation whose precondition is satisfied;
    (* there always is one *)
  od.
```

An execution of $relabel(v)$ is called a *relabeling* of v. Define a push to be *regular* if it does not take place during a call of *add edge*; there are at most m nonregular pushes.

Note the special status of the source and the sink: no regular pushes are performed out of s or t, nor are they ever relabeled.

We next show the partial correctness of the algorithm (i.e., if it terminates, it will do so with the correct result) and give a few additional properties. Our proof is similar to the correctness proof of the generic algorithm of [GT88]. In stating invariants for the algorithm, we consider *push*, *relabel*, and *add edge* to be atomic operations; i.e., we ignore possible violations of the invariants while these routines are being executed. We also implicitly restrict attention to the part of the execution that follows the initialization.

LEMMA 3.1. *At all times during an execution of the incremental generic algorithm and for all $v \in V^+$ if $d(v) > 0$ then $excess^*(v) \geq 0$.*

Proof. We use induction on the number of steps executed by the algorithm. The claim is clearly true immediately after the initialization and after a relabeling of v (by the precondition of the *relabel* operation). A push into v does not decrease the visible excess of v, and a regular push out of v does not decrease it below zero (by the precondition of the *push* operation). Finally, observe that the execution of an *add edge* operation cannot decrease the visible excess of any vertex. □

LEMMA 3.2. *At all times during the execution,*

(a) *f is a preflow,*

(b) *d is a valid labeling.*

Proof. (a) and (b) hold initially, and they are not invalidated by calls of *push* or *relabel* (cf. [GT88, Lem. 3.1]). Furthermore calls of *add edge* are easily seen to preserve (b). The only remaining issue is that for some $v \in V \backslash \{s\}$, a saturating push on an edge (v, w) performed during a call of *add edge* might invalidate the nonnegativity constraint $excess(v) \geq 0$. However, when the push takes place, $d(v) > d(w) \geq 0$, and it follows from Lemma 3.1 that $excess\ (v) \geq 0$ after the push. □

LEMMA 3.3. *Suppose that the algorithm terminates. Then, at termination, f is a maximum flow in G.*

Proof. At termination, $G^* = G$ and $excess(v) = excess^*(v) = 0$ for all $v \in V^+$. Thus f is a flow in G. If f is not maximum, then, by a classical theorem of Ford and Fulkerson [FF62, Cor. 5.2], there exists an augmenting path with respect to f, i.e., a simple path in G from s to t all of whose edges are residual with respect to f. Since $d(s) = n$, $d(t) = 0$, and the length of a simple path in G is at most $n - 1$, this contradicts the validity of d. □

LEMMA 3.4. *An ineligible edge $(v, w) \in E^*$ can become eligible only during a relabeling of v.*

Proof. An edge $e = (v, w)$ is ineligible exactly if either $rescap(e) = 0$ or $d(v) \leq d(w)$. The residual capacity of e can increase from zero to a positive value only during a push on $rev(e)$. But $d(w) > d(v)$ at the time of such a push on $rev(e)$; i.e., e is ineligible after the push. Thus only a relabeling of v can make e eligible. □

Lemmas 3.5 and 3.6 below are analogous to Lemmas 3.5 and 3.7 of [GT88], respectively. We include them for the sake of completeness.

LEMMA 3.5. *For all $v \in V^+$ and at all times during the execution, if $excess(v) > 0$, then there is a simple path in G^* from v to s all of whose edges are residual with respect to f.*

Proof. Let S be the set of vertices reachable from v in G^* by a path all of whose edges are residual with respect to f. We need to show that $s \in S$. Assume otherwise. The choice of S implies that $f(u, w) \leq 0$ for all edges $(u, w) \in E^*$ with $u \notin S$ and $w \in S$. Using the antisymmetry of f, we obtain $\sum_{w \in S} excess(w) =$

$\sum_{(u,w)\in E^*:u\in V,w\in S} f(u,w) = \sum_{(u,w)\in E^*:u\in V\setminus S,w\in S} f(u,w) + \sum_{(u,w)\in E^*:u\in S,w\in S} f(u,w) \le 0$. Since $excess(w) \ge 0$ for all $w \in V\setminus\{s\}$, we conclude that $excess(v) = 0$, a contradiction. □

LEMMA 3.6. *For all $v \in V$ and at all times during the execution, $d(v) \le 2n-1$. In particular, the total number of relabelings executed by the algorithm is $< 2n^2$.*

Proof. Let $v \in V^+$ with $excess(v) > 0$ be arbitrary. By Lemma 3.5 there is a simple path from v to s in G^* all of whose edges are residual with respect to f. Since $d(s) = n$, d is valid, and a simple path consists of at most $n-1$ edges, this implies that $d(v) \le 2n-1$. Thus no vertex v with $d(v) = 2n-1$ can ever be relabeled. □

We discuss instances of the incremental generic algorithm in §§4, 5, and 7. For all instances an efficient implementation of the following *current-edge abstract data type* is important: the task is to maintain two functions, $r : E \to \{0,1\}$ and $h : V \to \{0,\dots,2n-1\}$, under the operations specified below. An edge $(v,w) \in E$ is called *admissible* if $r(v,w) = 1$ and $h(v) = h(w)+1$. For $v \in V$, let $E(v) = \{(v,w) \in E : (v,w) \text{ is admissible}\}$.

Init;
Sets $h(v) := 0$ for all $v \in V\setminus\{s\}$, $h(s) := n$, and $r(v,w) := 0$ for all $(v,w) \in E$;

Spush(v,w);
Precondition: $v \in V$ and $(v,w) \in E(v)$.
Sets $r(v,w) := 0$ and $r(w,v) := 1$;

Npush(v,w);
Precondition: $v \in V$ and $(v,w) \in E(v)$.
Sets $r(v,w) := r(w,v) := 1$;

Relabel(v);
Precondition: $v \in V$, $E(v) = \emptyset$, and $h(v) < 2n-1$.
Executes $h(v) := h(v)+1$;

Add edge$(\{v,w\})$;
Precondition: $\{v,w\} \in \overline{E}$ and $r(v,w) = r(w,v) = 0$.
Sets $r(v,w) := 1$ if either $h(v) < h(w)$ or $(h(v) = h(w)$ and $cap(v,w) > 0)$;
Sets $r(w,v) := 1$ if either $h(w) < h(v)$ or $(h(v) = h(w)$ and $cap(w,v) > 0)$;

ce(v);
Precondition: $v \in V$.
Returns some $(v,w) \in E(v)$ if $E(v) \ne \emptyset$, *nil* otherwise;

For $q \in \mathbb{N}$, denote by $T_{ce}(n,m,q)$ the time needed to execute any legal sequence of one *Init* operation followed by q *Spush*, *Npush*, *Relabel*, *Add edge*, and *ce* operations. Note that such a sequence contains at most m *Add edge* operations, since at all times after a call *Add edge*$(\{v,w\})$ we have $r(v,w) = 1$ or $r(w,v) = 1$. Implementations of the current-edge data type are discussed in §§6 and 8. The algorithms of §§4 and 5 use the current-edge data type as follows: *generic initialize* calls *Init*; a saturating and a nonsaturating regular push on an edge e call *Spush*(e) and *Npush*(e), respectively; *relabel*(v) calls *Relabel*(v); and *add edge*$(\{v,w\})$ calls *Add edge*$(\{v,w\})$. For the sake of simplicity we do not explicitly mention these calls in the description of the algorithms, and we consider them to form atomic entities with the calling operations.

With this interface, it is easy to verify that the following holds throughout the execution: $h(v) = d(v)$ for all $v \in V$, and $r(v,w) = 1$ if and only if (v,w) is residual, for all $(v,w) \in E^*$. To see that the latter condition holds after a call *add edge*$(\{v,w\})$, recall that the call saturates (v, w) if $d(v) > d(w)$, saturates (w, v) if $d(w) > d(v)$, and leaves the flow on (v, w) at zero if $d(v) = d(w)$. Thus an edge $(v, w) \in E^*$ is eligible if and only if it is admissible, and for all $v \in V$ a call $ce(v)$ returns an eligible edge with tail v if there is one and *nil* otherwise. The function *ce* is used by the flow algorithms to find eligible edges on which to push flow and to test whether a vertex can be relabeled.

4. The incremental excess scaling algorithm.

In this section, we describe an incremental excess scaling algorithm for the case in which all edge capacities are integers bounded by $U \geq 1$. The algorithm is an adaptation of the excess scaling algorithm of Ahuja and Orlin [AO89] to the incremental paradigm.

The execution of the algorithm is divided into *phases* parameterized by the value of a *scaling parameter* Δ (of type *flow value*). The algorithm repeatedly chooses a vertex $v \in V^+$ with $excess^*(v) \geq \Delta$ and minimal $d(v)$ and either pushes flow on an edge (v, w) or relabels v. When there are no more vertices $v \in V^+$ with $excess^*(v) \geq \Delta$, the current phase ends, Δ is replaced by $\Delta/2$, all edges $(v, w) \in E \backslash E^*$ with $cap(\{v, w\}) \geq \Delta/\beta$ are added to E^*, and the next phase begins. Here β is a positive integer, which we will later fix at $\lfloor (m/n)^{1/2} \rfloor \geq 1$. The complete program follows.

```
INCREMENTAL EXCESS SCALING ALGORITHM:
  generic initialize;
  $L$ := list of all undirected edges in $\overline{E}$ ordered by decreasing capacities;
  $\Delta := 2^{\lfloor \log U \rfloor}$;
  while $\Delta \geq 1$
  do
    while $L \neq \emptyset$ and $cap(first(L)) \geq \Delta/\beta$ do add edge$(pop(L))$ od;
    while $\max\{excess^*(v) : v \in V^+\} \geq \Delta$
    do
      Among the vertices $v \in V^+$ with $excess^*(v) \geq \Delta$, choose $v$ as one with minimal
      $d(v)$;
      if $ce(v) = nil$
      then $relabel(v)$
      else $e := ce(v)$; $push(e, \min\{\Delta, rescap(e)\})$ fi;
    od;
    $\Delta := \Delta/2$;
  od.
```

If and when the algorithm terminates, we have $E^* = E$ (since $\beta \geq 1$) and *excess*$^*(v) < 1$ for all $v \in V^+$. Since all flow values computed by the algorithm are integral, this implies that $excess^*(v) = 0$ for all $v \in V^+$ at that point. Thus the algorithm is an instance of the incremental generic algorithm and hence is partially correct. The algorithm refines the excess scaling algorithm of Ahuja and Orlin [AO89]; in fact, for $\beta = \infty$, i.e., if all edges are added before the first phase, the two algorithms are identical.

Denote by $\sharp$*pushes*, $\sharp$*relabels*, $\sharp$*add edge*, and $\sharp$*ce* the total number of regular pushes, relabelings, calls to *add edge*, and calls to *ce*, respectively, executed by the algorithm. We first analyze $\sharp$*pushes* using a potential argument inspired by that of [CH89, Lem. 2]. Although part of this argument appears identically in [CH95], we repeat it in full for the reader's convenience. The argument is used to bound the number of regular saturating as well as the number of nonsaturating pushes. The analysis of the excess scaling algorithm by Ahuja and Orlin [AO89] also uses a potential argument. Their argument, however, applies only to nonsaturating pushes.

For $v \in V$ and $i = 1, 2, \ldots$, denote by $\deg_i(v)$ the number of edges with head v added to E^* between phase $i-1$ and phase i (for $i = 1$: before the first phase). Further, for $i = 1, 2, \ldots$, let $m_i = \sum_{v \in V} \deg_i(v)$.

Fact 4.1. At the time of a regular push on an edge $e = (v, w)$, e is eligible, the value of the push is $\le \Delta$, and if $w \in V^+$ then $excess^*(w) < \Delta$ immediately before the push.

LEMMA 4.1. *For all $v \in V^+$, $excess^*(v) < 2\Delta$ at the beginning of phase* 1, *and $excess^*(v) < 2\Delta + 2\deg_i(v)\Delta/\beta$ at the beginning of phase i for $i \ge 2$.*

Proof. Consider first the case $i = 1$. A call *add edge*$(\{s, v\})$ increases $excess^*(v)$ by at most $U < 2\Delta$, and there is at most one such call for each vertex $v \in V^+$. All other calls of *add edge* before phase 1 leave the flow unchanged since all vertices v except s have $d(v) = 0$. This completes the case $i = 1$. For $i \ge 2$ observe that $excess^*(v) < 2\Delta$ for all $v \in V^+$ before the calls of *add edge* between phases $i - 1$ and i and that each call *add edge*$(\{u, v\})$ between these phases adds at most $2\Delta/\beta$ to $excess^*(v)$. Thus $excess^*(v) < 2\Delta + 2\deg_i(v)\Delta/\beta$ at the beginning of phase i for all $i \ge 2$. □

For $\gamma \ge 1$, call a regular push on an edge (u, v) a *γ-push* if $|\{w \in V : d(w) = d(v)\}| \ge \gamma$ at the time of the push, and define the *γ-fooling height* $d_\gamma(v)$ of a vertex $v \in V$ as follows: if $V = \{v_1, \ldots, v_n\}$, then

$$d_\gamma(v) = \max_{i_1 \ge d(v_1), \ldots, i_n \ge d(v_n)} |\{k \in \mathbb{Z} : 0 \le k < d(v) \text{ and } |\{j : i_j = k\}| \ge \gamma\}|.$$

Intuitively, $d_\gamma(v)$ counts the maximum number of "dense virtual distance levels" between v and t, where a vertex v_j is allowed to occupy any one virtual distance level numbered at least $d(v_j)$ and where a dense virtual distance level is one that contains at least γ vertices.

d_γ has the following properties, named for future reference:

(F1) $\forall v \in V : 0 \le d_\gamma(v) \le n/\gamma$;

(F2) $\forall v \in V : d(v) = 0 \Rightarrow d_\gamma(v) = 0$;

(F3) $\forall u, v \in V : d(u) > d(v) \Rightarrow d_\gamma(u) \ge d_\gamma(v)$;

(F4) $\forall u, v \in V : (d(u) > d(v) \text{ and } |\{w \in V : d(w) = d(v)\}| \ge \gamma) \Rightarrow d_\gamma(u) > d_\gamma(v)$;

(F5) A relabeling of a vertex $v \in V^+$ increases $d_\gamma(v)$ by at most 1 and does not increase $d_\gamma(w)$ for any $w \in V \backslash \{v\}$.

Define the *normalized value* of a push as the value of the push divided by Δ.

LEMMA 4.2.

(a) *For all $\gamma \ge 1$ the total normalized value of all γ-pushes is at most $(2n^2 \log U + 2nm/\beta)/\gamma + 4n^2$;*

(b) *there are $O(nm/\beta + n^2(\log U + 1))$ nonsaturating pushes;*

(c) *there are $O(nm/\beta + n^2\beta + n^2 \log U)$ saturating pushes.*

Proof.

(a) Define the potential function

$$\Phi = \sum_{v \in V^+ : excess^*(v) \le 2\Delta} \frac{excess^*(v)}{\Delta} \cdot d_\gamma(v) + \sum_{v \in V^+ : excess^*(v) > 2\Delta} \frac{excess^*(v)}{\Delta} \cdot \frac{n}{\gamma}.$$

At the start of phase 1, $\Phi = 0$ (by Lemma 4.1, property (F2), and the fact that $d(v) = 0$ for all $v \in V^+$ at the start of phase 1), and $\Phi \ge 0$ always (by Lemma 3.1 and property (F2)). Φ does not increase due to regular pushes (by Fact 4.1 and properties (F1) and (F3)), and a relabeling increases Φ by at most 2 (by property (F5)). For $i \geq 2$, the change of Δ and the addition of edges between phases $i-1$ and i increase Φ by at most $(2n + 2m_i/\beta) \cdot n/\gamma$ (by Lemma 4.1 and property (F1)). Consequently, and by Lemma 3.6, the total increase, and hence also the total decrease, in Φ is at most $(2n^2 \log U + 2nm/\beta)/\gamma + 4n^2$. Finally, note that each γ-push of normalized value c causes Φ to decrease by at least c (by Fact 4.1 and property (F4)).

(b) Every push is a 1-push and every nonsaturating push has normalized value 1. The bound now follows from part (a), applied with $\gamma = 1$.

(c) Call a regular push on an edge (u, v) *small* if its value is less than Δ/β, call it *terminal* if $|\{w \in V : d(w) = d(v)\}| < \beta$ at the time of the push, and partition the regular saturating pushes into three classes: (1) small pushes, (2) nonsmall terminal pushes, and (3) nonsmall nonterminal pushes. We bound the number of pushes in each class separately.

(1) We have $cap(\{v, w\}) \geq \Delta/\beta$ for each $\{v, w\} \in \overline{E^*}$. Hence between any two small saturating pushes on a fixed undirected edge there is a nonsaturating push on that edge. Therefore the number of small saturating pushes is at most m plus the number of nonsaturating pushes, and the bound follows from part (b).

(2) By Lemma 3.4, each terminal push out of a vertex $v \in V$ is followed by fewer than β saturating pushes out of v before the next relabeling of v. Summing over all $v \in V$ and all possible values of $d(v)$, this gives at most $2n^2\beta$ terminal saturating pushes.

(3) The normalized value of a nonsmall push is at least $1/\beta$, and each nonterminal push is a β-push. An application of part (a) with $\gamma = \beta$ now shows that there are at most $2n^2 \log U + 2nm/\beta + 4n^2\beta$ regular nonsmall nonterminal pushes. □

We sum up the findings in the following theorem.

THEOREM 4.1. *A maximum flow in a network with n vertices, m edges, and integer capacities bounded by $U \geq 1$ can be computed deterministically using $O(q)$ flow operations and $O(q) + T_{ce}(n, m, q)$ time, where $q = O(n^{3/2}m^{1/2} + n^2 \log U)$.*

Proof. Put $\beta = \lfloor (m/n)^{1/2} \rfloor$ and note that β can be computed within the stated resources. Sorting the undirected edges by their capacities takes $O(m \log m) = O(n^{3/2}m^{1/2})$ time, the execution of *generic initialize* is no more expensive, and the initial value of Δ can be computed in $O(m + \log U)$ time. There are $\lfloor \log U \rfloor + 1$ phases, in each of which a number of undirected edges is added to $\overline{E^*}$. This takes $O(\log \beta) = O(\log n)$ time per undirected edge (for the multiplication of its capacity by β) and hence $O(m \log n)$ time altogether. Using simple data structures described in [AO89], the selection of v in the second inner while loop of the algorithm can be implemented to run in constant time per vertex selection plus $O(n)$ time per phase. Over the whole algorithm, this adds up to $O(\sharp pushes + \sharp relabels + n(\log U + 1))$ time. Since $\sharp pushes = O(n^{3/2}m^{1/2} + n^2 \log U)$ (by Lemma 4.2), $\sharp relabels = O(n^2)$ (by Lemma 3.6), $\sharp ce = O(\sharp pushes + \sharp relabels)$, and $\sharp add\ edge \leq m$, both the total number of operations executed on the current-edge data structure and the total time spent outside this data structure are $O(n^{3/2}m^{1/2} + n^2 \log U)$. The claim follows. □

In the next section, we show how the wave scaling technique of [AOT89] can be combined with the incremental approach to reduce the value of q in Theorem 4.1 to $O(n^{3/2}m^{1/2} + n^2(\log U)^{1/2} + \log U)$.

5. The incremental wave scaling algorithm.

The incremental wave scaling algorithm is an adaptation of the wave scaling algorithm of [AOT89] to the incremental paradigm. As in the previous section all edge capacities are integers bounded by

$U \geq 1$, and the execution is divided into phases parameterized by Δ, with edge additions taking place between phases.

The incremental wave scaling algorithm makes use of the procedures *stack push relabel* and *wave*. A call *stack push relabel*(v) pushes flow out of v until either the visible excess of v is zero or there are no eligible edges with tail v, in which case v is relabeled. Also, when *stack push relabel*(v) considers an eligible edge (v, w) and w has visible excess Δ or more, *stack push relabel* is first called recursively with argument w in order to "clear the way" for the push on (v, w). The procedure *wave* orders the vertices in V^+ by decreasing values of the functions d and then steps through the ordered list of vertices, calling *stack push relabel* for each vertex in turn. An important property of processing the vertices in this order is that once a call *stack push relabel*(v) in *wave* has terminated, the visible excess of v remains unchanged until the end of the call of *wave*. The algorithm employs the two procedures as follows: in each phase, it first uses *stack push relabel* to reduce the individual visible excess of each vertex in V^+ below Δ and then *wave* to reduce Σ^* below $n\Delta/l$, where $l > 0$ is a parameter to be chosen later and $\Sigma^* = \sum_{v \in V^+} \max\{excess^*(v), 0\}$. Although not strictly accurate, it is helpful to think of Σ^* as the total visible excess.

```
PROCEDURE stack push relabel(v: vertex);
  while $excess^*(v) > 0$ and $ce(v) \neq nil$
  do
    $w := head(ce(v))$;
    if $w \neq t$ and $excess^*(w) \geq \Delta$
    then stack push relabel(w)
    else (* $w = t$ or $excess^*(w) < \Delta$ *)
      $push((v, w), \min\{excess^*(v), \Delta, rescap(v, w)\})$;
    fi;
  od;
  if $excess^*(v) > 0$ then relabel(v) fi;

PROCEDURE wave;
  $J :=$ list of all vertices $v \in V^+$ ordered by decreasing values of $d(v)$;
  while $J \neq \emptyset$
  do stack push relabel(pop(J)) od;

INCREMENTAL WAVE SCALING ALGORITHM:
  generic initialize;
  $L :=$ list of the undirected edges in $\overline{E}$ ordered by decreasing capacities;
  $\Delta := 2^{\lfloor \log U \rfloor}$;
  while $\Delta \geq 1$
  do
    while $L \neq \emptyset$ and $cap(first(L)) \geq \Delta/\beta$ do add edge(pop(L)) od;
    (* A *)
    while $\max\{excess^*(v) : v \in V^+\} \geq \Delta$
    do
      Choose $v \in V^+$ with $excess^*(v) \geq \Delta$;
      stack push relabel(v);
    od;
    (* B *)
    while $\Sigma^* \geq n\Delta/l$ do wave od;
    (* C *)
    $\Delta := \Delta/2$;
od.
```

The parameters l and β will be chosen later. The incremental wave scaling algorithm differs in two respects from the wave scaling algorithm of [AOT89]: (1) it is incremental; and (2) instead of beginning each phase with a sequence of *waves*, i.e., calls of *wave*, we first reduce the visible excess of every vertex below Δ before executing the waves. This is necessary because the addition of edges at the beginning of a phase may cause individual excesses to be very large. In return, it is not necessary to reduce individual excesses after the waves as in [AOT89]—this is taken care of by the next phase.

We next elucidate the relationship between the incremental excess scaling algorithm of §4 and the incremental wave scaling algorithm of this section. Without the "wave loop," i.e., the loop between labels B and C, the two algorithms are basically the same. The wave loop reduces Σ^* below $n\Delta/l$. This allows us to replace the $n^2 \log U$ term in Lemma 4.2 by $n^2 \log U/l$ and thus yields an improved bound on the number of nonsaturating pushes. On the other hand, the wave loop brings about additional cost proportional to $n^2 l$ (since each wave has cost $\Theta(n)$ that cannot be accounted for by the techniques of the previous section, and since the number of waves is essentially proportional to nl; cf. Lemma 5.2). Choosing $l = (\log U)^{1/2}$ balances the two contributions and reduces the $n^2 \log U$ term in Theorem 4.1 to $n^2 (\log U)^{1/2}$ in Theorem 5.1.

We now analyze the incremental wave scaling algorithm. If and when the algorithm terminates, we have $E^* = E$ (since $\beta \geq 1$) and *excess*$^*(v) < 1$ for all $v \in V^+$. Thus the algorithm is an instance of the incremental generic algorithm and therefore partially correct. Also, Fact 4.1 holds for it. Lemma 4.1 can be sharpened to the following.

Fact 5.1. For $i \geq 1$, the following bounds on total and individual visible excesses hold during phase i:

(a) at label A, *excess*$^*(v) < 2\Delta$ for $i = 1$ and for all $v \in V^+$, and $\Sigma^* < 2n\Delta/l + 2m_i\Delta/\beta$ for $i > 1$;

(b) at label B and until the end of the phase, *excess*$^*(v) < 2\Delta$ for all $v \in V^+$.

Denote by $\sharp$*stack push relabels* and by $\sharp$*waves* the number of calls of *stack push relabel* and of *wave*, respectively, and by $\sharp$*pushes* the number of regular pushes executed by the algorithm. We first show the following refinement of Lemma 4.2.

LEMMA 5.1.

(a) *For all $\gamma \geq 1$, the total normalized value of all γ-pushes is at most* $2n^2 \log U/(l\gamma) + 2nm/(\beta\gamma) + 4n^2$;

(b) $\sharp$*stack push relabels* $= O(nm/\beta + n^2 + n^2 \log U/l + n \cdot \sharp$*waves*$)$;

(c) $\sharp$*pushes* $= O(nm/\beta + n^2\beta + n^2 \log U/l + n \cdot \sharp$*waves*$)$.

Proof.

(a) We use the same potential function Φ as in the proof of Lemma 4.2(a). At the start of phase 1, $\Phi = 0$ (by Fact 5.1(a), property (F2) of γ-fooling height, and the fact that $d(v) = 0$ for all $v \in V^+$ at the beginning of phase 1). By the same argument as in the proof of Lemma 4.2(a), $\Phi \geq 0$ always, Φ does not increase due to regular pushes, and a relabeling increases Φ by at most 2. For $i \geq 2$, the change of Δ and the addition of edges between phases $i-1$ and i increase Φ by at most $(2n/l + 2m_i/\beta) \cdot n/\gamma$ (by Fact 5.1(a) and property (F1)). Consequently, the total decrease in Φ is at most $2(n/l) \cdot (n/\gamma) \cdot \log U + 2(m/\beta) \cdot (n/\gamma) + 4n^2$. Finally, note that each γ-push of normalized value c causes Φ to decrease by at least c (by property (F4)).

(b) Define a call *stack push relabel*(v) to be *potent* if *excess*$^*(v) \geq \Delta$ at the time of the call. Since each nonpotent call of *stack push relabel* is made directly by *wave*, there can be at most $n \cdot \sharp$*waves* such calls. Also at most $2n^2$ calls *stack push relabel*(v) end with a relabeling of v (by Lemma 3.6). A potent call *stack push relabel*(v) that does not end with a relabeling of v, finally, carries out pushes out of v of total normalized value at least 1. Since every push is a 1-push, an application of part (a) with $\gamma = 1$ now shows the number of such calls to be $O(nm/\beta + n^2 + n^2 \log U/l)$.

(c) There is at most one nonsaturating push of value $< \Delta$ per call of *stack push relabel*, and the number of pushes of value $\geq \Delta$ is easily bounded by another application of part (a) with $\gamma = 1$. This shows the bound for nonsaturating pushes. As for saturating pushes, we define the concepts of small and terminal pushes as in the proof of Lemma 4.2(c) and argue as was done there. The number of small saturating pushes is bounded by m plus the number of nonsaturating pushes, there are $O(n^2\beta)$ terminal pushes, and the number of nonsmall nonterminal pushes is $O(n^2 \log U/l + nm/\beta + n^2\beta)$. □

The following lemma was essentially proved in [AOT89] (Lemma 4.2).

LEMMA 5.2. $\sharp$*waves* $= O(\min\{n^2, nl + \log U\})$.

Proof. We first show the $O(n^2)$ bound and then the $O(nl + \log U)$ bound.

For the $O(n^2)$ bound, observe first that at most $2n^2$ waves execute a relabeling (Lemma 3.6). On the other hand, a wave that does not execute at least one relabeling reduces Σ^* to zero and hence is either the last wave or is separated from the next wave by the addition to E^* of at least one edge. Thus there are at most $2n^2+m+1 = O(n^2)$ waves.

For the $O(nl + \log U)$ bound, consider any wave that is not the last in its phase. At the end of such a wave $\Sigma^* \geq n\Delta/l$. Also, no vertex has visible excess exceeding 2Δ (by Fact 5.1(b)), and every vertex with positive visible excess at the end of the wave was relabeled during the wave. Thus at least $n/(2l)$ relabelings occurred during the wave and hence the number of waves is bounded by $\lfloor \log U \rfloor + 1 + 2n^2/(n/(2l)) = O(nl + \log U)$. □

THEOREM 5.1. *A maximum flow in a network with n vertices, m edges, and integer capacities bounded by $U \geq 1$ can be computed deterministically using $O(q + \log U)$ flow operations and $O(q + \log U) + T_{ce}(n, m, q)$ time, where $q = O(n^{3/2}m^{1/2} + n^2(\log U)^{1/2})$.*

Proof. Replace U by $\max\{U, 2\}$ and choose l such that $l = \Theta(\sqrt{\log U})$ and such that the sequence $a_0, a_1, \ldots, a_{\lfloor \log U \rfloor}$ can be computed in $O(\log U)$ time, where $a_i = \lceil 2^i n/l \rceil$ for $i \geq 0$. We show below how to do this. Also take $\beta = \lfloor (m/n)^{1/2} \rfloor$ (as in the previous section) and note that β can be computed within the stated resources. As in the proof of Theorem 4.1, a component in the running time of $O(m \log n + \log U)$ accounts for initialization, maintenance of Δ, and multiplication of the capacities of all undirected edges by β. By the conditions placed on l above, each execution of the test between labels B and C can be carried out in constant time (maintain Σ^* explicitly). The total number of operations executed on the current-edge data structure as well as the remaining running time can be seen to be at most proportional to $\sharp$*pushes* + $\sharp$*stack push relabels* + m; in particular, note that the list J employed by *wave* can be constructed in $O(n)$ time by bucket sorting. Since $\sharp$*waves* $= O(\min\{n^2, nl + \log U\}) = O(n(\log U)^{1/2})$, Lemma 5.1 implies that $\sharp$*pushes*+$\sharp$*stack push relabels* $= O(n^{3/2}m^{1/2} + n^2(\log U)^{1/2})$, from which the desired result follows.

In the remainder of the proof, we show how to choose l to satisfy the conditions stated above. This material can be skipped in a first reading, and it is of little relevance to readers with no interest in the details of our model of computation.

What makes the problem nontrivial is the insistence of our model on a neat separation between the types *integer* and *flow value* and the very restricted operations applicable to values of type *flow value*. For example, the condition $U > n$ cannot be tested directly because of a type mismatch: U is of type *flow value*, while n is of type *integer*, and we have not provided for comparisons between values of different types. The available operations do, however, allow the computation of the *flow value* n from the *integer* n in $O(n)$ time (and, in fact, in $O(\log n)$ time), so that the test can be executed after all. We use this idea below.

Begin by computing $\lceil \log U \rceil = \min\{i \geq 1 : 2^i \geq U\}$ in $O(\log U)$ time using repeated doubling. Then determine $\lceil \sqrt{\log U} \rceil = \min\{i \geq 1 : \sum_{j=1}^{i}(2j-1) \geq \lceil \log U \rceil\}$ in $O(\sqrt{\log U})$ time. Finally compute $i_0 = \min\{i \in \mathbb{Z} : 2^i n \geq \lceil \sqrt{\log U} \rceil\}$ and take $l = 2^{i_0} n$. The numbers i_0 and $2^{|i_0|}$ can be found in $O(|i_0|+1) = O(\log(n+\log U))$ time via repeated doubling, starting from $\min\{n, \lceil \sqrt{\log U} \rceil\}$, and clearly $l = \Theta(\sqrt{\log U})$. Furthermore, $a_i = \lceil 2^{i-i_0} \rceil$ for $i \geq 0$. Hence a_0 can be computed in constant time, and a_i can be computed from a_{i-1} in constant time for all $i \geq 1$. □

6. Solutions to the current-edge problem.

In this section we describe two solutions to the current-edge problem, i.e., implementations of the current-edge data type, both of which are based on the fact that if an edge $(v, w) \in E^*$ is inadmissible at some time then it remains inadmissible until the next execution of *Relabel*(v) (cf. Lemma 3.4).

LEMMA 6.1. $T_{ce}(n, m, q) = O(nm + q) = O(n^3 + q)$.

Proof. Maintain for each vertex v a list L_v containing those edges (v, w) for which *Add edge*$(\{v, w\})$ has been executed. If the functions r and h are represented by tables in the obvious way, *Init* can be executed in $O(m)$ time, while each of *Spush*, *Npush*, *Relabel*, and *Add edge* takes constant time. In order to implement the final operation *ce*, additionally maintain for each vertex v a pointer $z[v]$ into L_v which is initialized to point to the beginning of L_v and is reset to this position in each call of *Relabel*(v). Each call *ce*(v) advances $z[v]$ (possibly a distance of zero) until an admissible edge is encountered, or until the end of L_v is reached, in which case the value *nil* is returned. The correctness of this implementation follows from the fact cited above, which guarantees that no edge behind $z[v]$ is ever admissible; in particular, note that an edge (v, w) is always inadmissible at the time of its insertion in L_v. Since each pointer $z[v]$ makes at most $2n$ scans over its list L_v, the total time spent is $O(nm + q)$. □

Lemma 6.1 describes the standard solution to the current-edge problem introduced in [GT88] and also used in [AO89], [AOT89], and [CH95]. We now give a faster solution. First, identify V with the set $\{0, \ldots, n-1\}$ and extend r to a function from $V \times V$ to $\{0, 1\}$ by taking $r(v, w) = 0$ for $(v, w) \notin E$.

THEOREM 6.1. $T_{ce}(n, m, q) = O(n^3/\log n + q)$.

Proof. We represent the function h not only directly but also through an array $H : \{0, \ldots, 2n-1\} \times V \to \{0, 1\}$ such that for all integers k with $0 \leq k \leq 2n-1$ and all $v \in V$, $H[k, v] = 1$ if and only if $h(v) = k$. Then for all $(v, w) \in E$ with $h(v) > 0$, $r(v, w) \cdot H[h(v)-1, w] \neq 0$ iff the edge (v, w) is admissible. We combine this observation with the "four Russians' trick" (see [AHU74, §6.6]); i.e., we partition each row of the arrays r and H into *blocks* of size x and represent the x bits of each block by a single integer. Here x is a positive integer, which for simplicity we assume to be a divisor of n. More precisely, let $X = \{0, \ldots, 2^x - 1\}$. Instead of r and H, we maintain arrays $r' : V \times \{0, \ldots, n/x - 1\} \to X$ and $H' : \{0, \ldots, 2n-1\} \times \{0, \ldots, n/x - 1\} \to X$ defined as follows: for all $v \in V$ and all integers k and i with $0 \leq k \leq 2n-1$ and $0 \leq i \leq n/x - 1$, let

$$r'[v,i] = \sum_{j=0}^{x-1} r(v, ix+j) \cdot 2^{x-1-j} \quad \text{and} \quad H'[k,i] = \sum_{j=0}^{x-1} H[k, ix+j] \cdot 2^{x-1-j}.$$

For $a \in X$ let $a^{(x-1)}, \ldots, a^{(0)}$ denote the individual bits of a, i.e., $a^{(x-1)}, \ldots, a^{(0)} \in \{0,1\}$ and $\sum_{j=0}^{x-1} a^{(j)} \cdot 2^j = a$. For $a, b \in X$ let $a \wedge b$ be the bitwise AND of a and b; i.e., $a \wedge b = \sum_{j=0}^{x-1} (a^{(j)} \cdot b^{(j)}) \cdot 2^j$. Then for all $v \in V$ with $h(v) > 0$ and for all integers i with $0 \leq i \leq n/x - 1$, we have $r'[v,i] \wedge H'[h(v) - 1, i] \neq 0$ iff one of the edges in $\{(v, ix+j) : 0 \leq j < x\}$ is admissible. This leads to the implementation of *ce* given below; the remaining operations are left to the reader. In order to understand the last line of the code, note that for each nonzero $a \in X$, $\lfloor \log a \rfloor$ is the position of the leftmost nonzero bit in a, the rightmost bit position counted as zero.

```
FUNCTION ce(v: vertex): edge;
  if $h(v) = 0$ then return nil fi;
  while $r'[v, z[v]] \wedge H'[h(v) - 1, z[v]] = 0$ and $z[v] < n/x - 1$
  do $z[v] := z[v] + 1$ od;
  if $r'[v, z[v]] \wedge H'[h(v) - 1, z[v]] = 0$
  then return nil
  else return $(v, z[v] \cdot x + x - 1 - \lfloor \log(r'[v, z[v]] \wedge H'[h(v) - 1, z[v]]) \rfloor)$ fi;
```

In the execution of any legal sequence of q operations following *Init*, the total number of changes to $z[v]$ is $O(n^2/x)$ for arbitrary $v \in V$. Hence such a sequence can be executed in $O(n^3/x + q)$ time, provided that the operations of testing and setting individual bits of numbers in X and of computing $\lfloor \log a \rfloor$ and $a \wedge b$ for arbitrary $a, b \in X$ take constant time.

For $x = \lfloor \log n \rfloor$, tables implementing the operations $a \mapsto \lfloor \log a \rfloor$ and $(a, b) \mapsto a \wedge b$ for $a, b \in X$ can be constructed in $O(n^2)$ time, and individual bits of numbers in X can be inspected and modified in constant time via appropriate multiplications and integer divisions by powers of two. This completes the proof of Theorem 6.1. $\Box$

Remark. On many real computers, the operation of bitwise AND is built in, i.e., takes constant time. It is easy to improve Theorem 6.1 for such a machine with a nonstandard word length of $x = \omega(\log n)$ bits. Although the remaining bit-level operations discussed above may not be available at unit cost, they can trivially be executed in $O(x)$ time; hence on a machine with a word length of x bits and unit-time bitwise AND, $T_{ce}(n, m, q) = O(n^3/x + qx + n^2)$, where the term n^2 accounts for the cost of initialization.

7. The incremental strongly polynomial algorithm.

In addition to the data structures of the generic algorithm, the incremental strongly polynomial algorithm uses, as do several previous algorithms, an edge-weighted directed graph $F = (V, E_F, val)$, where $E_F \subseteq E^*$ and *val* is a function from E_F to $\mathbb{R}$. F at all times is a directed forest, i.e., an acyclic directed graph with maximum outdegree at most one, and $val(e) = rescap(e)$ for all $e \in E_F$. A vertex $v \in V$ is called a *root* exactly if its outdegree in F is zero. The following operations are applied to F:

InitF;
Sets $E_F := \emptyset$;

Find value(e);
Precondition: $e \in E_F$.
Returns *val*(e);

Find root(v);
Precondition: $v \in V$.
Returns the root of the tree in F containing v;

Find min(v);
Precondition: $v \in V$ and v is not a root.
Returns an edge e of minimal value *val*(e) on the maximal path in F starting at v; in the case of ties the last such edge is returned;

Add value(v, c);
Precondition: $v \in V$ and $c \in \mathbb{R}$.
Replaces *val*(e) by *val*$(e) + c$ for each edge e on the maximal path in F starting at v;

Link(e, c);
Precondition: $e \in E^*$, $c \in \mathbb{R}$, and $(V, E_F \cup \{e\})$ is a directed forest.
Replaces E_F by $E_F \cup \{e\}$ and sets *val*$(e) := c$;

Cut(e);
Precondition: $e \in E_F$.
Replaces E_F by $E_F \backslash \{e\}$;

The *dynamic trees* data structure of Sleator and Tarjan [ST85] supports the seven operations defined above in $O(\log n)$ amortized time each; i.e., a sequence of q operations on F, starting with *InitF*, can be executed in $O(q \log n)$ time.

The preflow f is represented in one of two ways: for $e \in E^*$, while $e \notin E_F$ and *rev*$(e) \notin E_F$, $f(e)$ is stored directly as $g[e]$, where $g : E \to \mathbb{R}$ is an array. While $e \in E_F$, $f(e)$ is given implicitly as *cap*$(e) -$ *val*(e) and $f($*rev*$(e))$ as $-f(e)$. Accordingly, we redefine the basic procedure *setflow* and incorporate the conventions for the representation of f into new versions of *Link* and *Cut*.

```
PROCEDURE setflow(e: edge; c: real);
  $g[e] := c$; $g[rev(e)] := -c$;

PROCEDURE link(e: edge);
  Link(e, rescap(e));

PROCEDURE cut(e: edge);
  setflow(e, cap(e) − Find value(e));
  Cut(e);
```

The procedure *tree push* defined below works as follows: a call *tree push*(v) first inserts an eligible edge with tail v into E_F if v is a root, and then determines the minimal residual capacity c of any edge on the maximal path in E_F starting at v. It finally increases the flow along that path by $\min\{c, excess^*(v)\}$ and deletes all edges from E_F that become saturated.

```
PROCEDURE tree push(v: vertex);
  if Find root(v) = v (* v is a root *) then link(ce(v)) fi;
  c := Find value(Find min(v));
  Add value(v, − $\min\{c, excess^*(v)\}$);
  while Find root(v) ≠ v and Find value(Find min(v)) = 0
  do cut(Find min(v)) od;
```

We finally extend the routine *relabel* and give the main program.

```
PROCEDURE relabel(v: vertex);
  for all $u \in V$ with $(u,v) \in E_F$ do cut(u, v) od;
  $d(v) := d(v) + 1$;

INCREMENTAL STRONGLY POLYNOMIAL ALGORITHM:
  generic initialize;
  InitF;
  $L :=$ list of the undirected edges in $\overline{E}$ ordered by decreasing capacities;
  while $L \neq \emptyset$
  do
    $\Delta := cap(first(L))$;        (* $\Delta$ is used only by the analysis *)
    add edge(pop(L));
    while $\max\{excess^*(v) : v \in V^+\} > 0$
    do
      Choose $v \in V^+$ with $excess^*(v) > 0$;
      if $ce(v) = nil$
      then relabel(v)
      else tree push(v);
      fi;
    od;
  od.
```

The algorithm uses the current-edge data type as follows: *generic initialize* calls *Init*, *relabel*(v) calls *Relabel*(v), *add edge*$(\{v,w\})$ calls *Add edge*$(\{v,w\})$, *link*(e) calls *Npush*(e), and each call *cut*(e) within *tree push* calls *Spush*(e). As will be seen below, the latter conventions regarding calls of *Npush* and *Spush* guarantee the invariant that each edge in E^* is eligible iff it is admissible, which in turn ensures the correctness of the values returned by calls of *ce*. The argument given in §3 for the equivalence of eligibility and admissibility no longer suffices. While clearly $h(v) = d(v)$ continues to hold for all $v \in V$, the fact that an edge $(v,w) \in E^*$ is residual iff $r(v,w) = 1$ is demonstrated in part (b) of Lemma 7.1.

LEMMA 7.1. *At all times of the execution after the initialization and except during calls of add edge, tree push, and relabel, the following invariants hold:*

(a) *every edge in E_F is eligible*;

(b) *for all $(v,w) \in E^*$, (v,w) is residual iff* $r(v,w) = 1$.

Proof. (a) and (b) hold vacuously immediately after the initialization (at which point $E_F = E^* = \emptyset$). Let us therefore assume, by way of induction, that they hold prior to a call of *add edge*, *tree push*, or *relabel*. We show that they hold after the call.

Invariant (a) can only be violated by a step that inserts an edge in E_F or that makes an edge in E_F ineligible. When an edge is inserted in E_F (in the call *link*$(ce(v))$), it was returned by *ce* immediately prior to the operation and hence is admissible at the time of the insertion. By invariant (b), it is also eligible at that time, so that invariant (a) is preserved. Note also that since traversing an eligible edge is always accompanied by a decrease in d value, the new edge does not form a cycle with the edges already in E_F; i.e., the precondition of the call of *Link* is satisfied. An edge in E_F becomes ineligible either because it is saturated, in which case it is removed from E_F in a call of *tree push*, or because of a relabeling, in which case it is removed from E_F in a call of *relabel*. In either case, since the edge no longer belongs to E_F, invariant (a) is preserved.

We now turn to invariant (b). Recall first from §3 that a call *Add edge*$(\{v,w\})$ initializes $r(v,w)$ and $r(w,v)$ correctly, i.e., in accordance with the invariant. When an edge (v,w) is inserted in E_F, a call *Npush*(v,w) sets $r(v,w)=1$, and $r(v,w)=1$ remains true until the call *Spush*(v,w), executed when (v,w) is saturated and removed from E_F (if this ever happens), which sets $r(v,w)=0$. Thus, by invariant (a), invariant (b) holds for all edges in E_F. When an edge (w,v) is inserted in E_F in a call of *tree push*, $r(v,w)$ is also set to 1. Furthermore, by invariant (a), the value of c computed by the call of *tree push* is strictly positive, so that a positive amount of flow is sent over (w,v) in the same call, making (v,w) residual if it were not so already. It is easy to see that as long as (w,v) remains in E_F, $r(v,w)$ remains equal to 1, and (v,w) remains residual. Invariant (b) therefore holds also for edges (v,w) such that (w,v) is in E_F. Since flow is pushed only over edges in E_F, while only insertions into and deletions from E_F change values of r, invariant (b) clearly holds for the remaining edges as well. $\square$

As an instance of the incremental generic algorithm, the incremental strongly polynomial algorithm is partially correct. The following fact is obvious.

Fact 7.1. At all times during the execution, $excess^*(v)\le\Delta$ for all vertices $v\in V^+$.

Define a *PTR* (*premature target relabeling*) *event* on an edge e to be a relabeling of the head of e while e is in E_F, and denote by $\sharp ptr$ the total number of PTR events during the execution. PTR events were introduced in [CH89], although with a somewhat different meaning. Their number depends on the exact edges returned by calls of *ce*; this dependence will be discussed in §8. A PTR event on an undirected edge $\{v,w\}$ is a PTR event on one of the edges (v,w) or (w,v). Denote by $\sharp$*sat cuts* the number of calls of *cut* within *tree push*, by $\sharp$*cuts* the sum of $\sharp$*sat cuts* and $\sharp ptr$ (observe that $\sharp ptr$ is the number of calls of *cut* within *relabel*), by $\sharp$*tree pushes* the number of calls of *tree push*, and by $\sharp$*links* the number of calls of *link*. A call of *cut* or *link* is also called a *cut* or a *link*, respectively.

The analysis of our strongly polynomial algorithm centers around the following ideas. We first show that the running time is determined by the search for current edges, $\sharp$*tree pushes*, and $\sharp$*cuts* and then relate $\sharp$*tree pushes* to $\sharp$*cuts*. In order to bound $\sharp$*sat cuts*, we use a potential function similar to the one used in the proof of Lemma 4.2, and in order to bound $\sharp ptr$, we use the analysis of [CH89].

LEMMA 7.2. *The algorithm uses* $O(q\log n)$ *flow operations and* $O((\sharp\textit{tree pushes}+\sharp\textit{cuts})\cdot\log n+n^2+m\log n)+T_{ce}(n,m,q)$ *time, where* $q=O(\sharp\textit{tree pushes}+\sharp\textit{cuts}+n^2)$.

Proof. It takes time $O(m\log n)$ to sort the undirected edges by capacity. If we maintain for each vertex v the set of edges in E_F with head v, then a relabeling takes $O(1)$ time plus $O(\log n)$ time for each cut caused by the relabeling. A call of *tree push* takes $O(\log n)$ time plus $O(\log n)$ time for each cut caused by *tree push*. Finally, the time spent on the current-edge task is $T_{ce}(n,m,q)$, where $q=O(\sharp\textit{tree pushes}+\sharp\textit{cuts}+n^2)$, since the number of calls of *ce* is bounded by the number of relabelings plus twice the number of calls of *tree push*, the numbers of calls of *Npush* and *Spush* are bounded by $\sharp$*tree pushes* and $\sharp$*cuts*, respectively, and there are $O(n^2)$ calls of *Add edge* and *Relabel*. $\square$

LEMMA 7.3.

(a) $\sharp\textit{tree pushes}=O(\sharp\textit{links}+\sharp\textit{sat cuts}+m)$;

(b) $\sharp\textit{links}\le\sharp\textit{cuts}+n$.

Proof. (a) We use a potential function Φ defined as the number of nonroot vertices with positive visible excess. When *tree push*(v) is called, we have *excess*$^*(v) > 0$. If a call *tree push*(v) performs neither a link nor a cut, then v was not a root before the call and *excess*$^*(v) = 0$ after the call; i.e., Φ is reduced by one. No call of *tree push* increases Φ by more than one, the addition of an undirected edge increases Φ by at most two, and a relabeling does not change Φ. Hence the total increase in Φ is $O(\sharp links + \sharp sat\ cuts + m)$, $\Phi = 0$ initially, and $\Phi \geq 0$ always, and, with the exception of at most $\sharp links + \sharp sat\ cuts$ calls, every call of *tree push* decreases Φ by one.

(b) Since F is a forest at all times during the execution, it never contains more than $n - 1$ edges. □

LEMMA 7.4. *$\sharp sat\ cuts = O(n^{3/2}m^{1/2} + \sharp ptr)$.*

Proof. Define a *push bundle* for an edge $e \in E^*$ to be the sequence of all regular pushes on e in a maximal period of time in which e belongs to E_F. A push bundle for an undirected edge $\{v, w\}$ is a push bundle for one of the edges (v, w) or (w, v). The number of push bundles clearly bounds $\sharp sat\ cuts$. A push bundle for an edge $e = (v, w) \in E^*$ is called *complete* if its pushes increase $f(e)$ by *cap*$(\{v, w\})$, i.e., from $-$*cap*(w, v) to *cap*(v, w), and *incomplete* otherwise.

CLAIM 1. *The number of incomplete push bundles is $O(m + \sharp ptr)$.*

Proof. A maximal period of time in which an edge e belongs to E_F is terminated by a PTR event on e, by a saturating push on e, or by the end of the execution. Hence an incomplete push bundle for an undirected edge $\{v, w\}$ that is neither the first nor the last push bundle for $\{v, w\}$ is either immediately preceded or immediately followed by a PTR event on $\{v, w\}$.

CLAIM 2. *The number of complete push bundles is $O(n^{3/2}m^{1/2})$.*

Proof. Define the *level* of a push on an edge $(u, v) \in E^*$ to be the value of $d(u)$ at the time of the push. Two pushes on a fixed edge (u, v) have the same level if and only if they belong to the same push bundle. Hence for all $(u, v) \in E$ and all integers k with $1 \leq k \leq 2n - 1$, we can denote by $\langle u, v, k\rangle$ the push bundle for (u, v) (if any) whose pushes are of level k. Let β be a positive integer, to be chosen below, and call a push bundle $\langle u, v, k\rangle$ *terminal* if it is followed by fewer than β push bundles of the form $\langle u, w, k\rangle$, where $(u, w) \in E$. Clearly, there are at most $2n^2\beta$ terminal push bundles. In order to count the number of nonterminal push bundles, we use a potential argument similar to those used in the proofs of Lemmas 4.2 and 5.1.

Consider the potential function

$$\Phi = \sum_{v \in V^+} \frac{excess^*(v)}{\Delta} \cdot d_\beta(v),$$

where d_β denotes the β-fooling height introduced in §4. $\Phi = 0$ initially and $\Phi \geq 0$ always (by Lemma 3.1 and property (F2) of β-fooling height), Φ does not increase due to changes of Δ (since *excess*$^*(v) \leq 0$ for all $v \in V^+$ at each change of Δ), Φ does not increase due to regular pushes (by property (F3)), the total increase due to relabelings is at most $2n^2$ (by Fact 7.1 and property (F5)), and the increase due to the addition of an undirected edge is at most n/β (by property (F1)). The total decrease of Φ is therefore bounded by $nm/\beta + 2n^2$. Finally, note that the pushes in a complete nonterminal push bundle decrease Φ by at least one. This can be seen as follows. Consider a complete nonterminal push bundle $\langle u, v, k\rangle$. Since $\langle u, v, k\rangle$ is nonterminal, it is followed by β bundles $\langle u, w_1, k\rangle, \ldots, \langle u, w_\beta, k\rangle$. Lemma 3.4 now implies that the edges $(u, w_1), \ldots, (u, w_\beta)$ are eligible whenever a push in the bundle $\langle u, v, k\rangle$ occurs. Thus, by property (F4), $d_\beta(u) > d_\beta(v)$ at the time of each such push. Also, the total value of the pushes in the push bundle is $cap(\{u,v\})$ and $\Delta \le cap(\{u,v\})$ whenever a push in the bundle occurs since the undirected edges are added in the order of decreasing capacities. Summing up, the total number of complete push bundles is $O(nm/\beta + n^2\beta)$. Claim 2 follows with $\beta = \lfloor (m/n)^{1/2} \rfloor$, and this ends the proof of Lemma 7.4. □

LEMMA 7.5. *The algorithm uses $O(q\log n)$ flow operations and $O(q \log n) + T_{ce}(n,m,q)$ time, where $q = O(n^{3/2}m^{1/2} + \sharp ptr)$.*

Proof. Combine Lemmas 7.2, 7.3, and 7.4. □

8. The extended current-edge problem and PTR events.

In the definition of the current-edge data type in §3, we allowed a call $ce(v)$ to return an arbitrary admissible edge (if any) with tail v. Given so much freedom, however, an adversary might be able to "score" a high number of PTR events, leading to a bad running time. Our defense against the adversary will be randomness: we force the choice among several admissible edges to be made according to a fixed but random ordering; then we can prove that the number of PTR events is usually much lower than the naive upper bound. In this section, we adapt the specification of the current-edge data type and extend the results of §6 to the more restrictive definition of *ce*, review and slightly extend the bounds on the number of PTR events shown in [CH89], and finally prove our main theorem.

For every finite set A, denote by Perm(A) the set of all permutations of A, i.e., of all bijections from $\{0,\dots,|A|-1\}$ to A. As in §6, identify V with the set $\{0,\dots,n-1\}$. The *extended current-edge data type* is initialized with n permutations $\xi_0, \xi_1, \dots, \xi_{n-1}$ of V. Its task is to maintain two functions $r : E \to \{0,1\}$ and $h : V \to \{0,\dots,2n-1\}$ under the operations *Init*, *Spush*, *Npush*, *Relabel*, *Add edge*, and *ce*. The operations *Spush*, *Npush*, *Relabel*, and *Add edge* are defined as in §3, and *Init* and *ce* are redefined as follows:

Init$(\xi_0,\dots,\xi_{n-1})$;
Precondition: $\xi_0,\dots,\xi_{n-1}$ are permutations of V.
Records $\xi_0,\dots,\xi_{n-1}$ and sets $h(v) := 0$ for $v \in V\backslash\{s\}$, $h(s) := n$, and $r(v,w) := 0$ for all $(v,w) \in E$.

ce(v);
Precondition: $v \in V$.
Returns the first admissible edge with tail v in the order induced by ξ_v if $E(v) \ne \emptyset$, *nil* otherwise.
If $E(v) \ne \emptyset$, the first admissible edge with tail v in the order induced by ξ_v is $(v, \xi_v(i_0))$, where $i_0 = \min\{i : 0 \le i \le n-1 \text{ and } (v,\xi_v(i)) \in E(v)\}$.

For $q \in \mathbb{N}$, denote by $T'_{ce}(n,m,q)$ the time needed to execute any legal sequence of one *Init* operation followed by q *Spush*, *Npush*, *Relabel*, *Add edge*, and *ce* operations of the extended current-edge data type.

LEMMA 8.1. $T'_{ce}(n,m,q) = O(nm+q) = O(n^3+q)$.

Proof. The proof is identical to that of Lemma 6.1, except that the list L_v is kept sorted according to the order induced by ξ_v for all $v \in V$. This makes *Add edge* operations more time-consuming. Since there are at most m calls of *Add edge*, however, each of which can be executed in $O(n)$ time, the total time is still $O(nm+q)$. □

We now extend the faster solution of §6, but only for a restricted class of permutations $\xi_0,\dots,\xi_{n-1}$. Let $x = \lfloor \log n \rfloor$, which, as in §6, we assume to be a divisor of n. Also, take $M = \{0,\ldots,n/x-1\}$ and let $B_i = \{ix, ix+1,\ldots,(i+1)x-1\}$, for $i = 0,\ldots,n/x-1$. A permutation of M is called a *block permutation.* For every block permutation $\Xi \in \mathrm{Perm}(M)$, define the *induced block-preserving permutation* as the permutation $\xi \in \mathrm{Perm}(V)$ obtained by first arranging the blocks according to Ξ and then replacing each block by the sorted sequence of its elements (i.e., for $v \in B_i$ and $w \in B_j$, $\xi^{-1}(v) < \xi^{-1}(w) \iff (\Xi^{-1}(i) < \Xi^{-1}(j)$ or $(i = j$ and $v < w)))$.

LEMMA 8.2. *For all $q \in \mathbb{N}$ and for n arbitrary block permutations $\Xi_0,\ldots,\Xi_{n-1} \in \mathrm{Perm}(M)$ with induced block-preserving permutations $\xi_0,\ldots,\xi_{n-1} \in \mathrm{Perm}(V)$, the operation Init$(\xi_0,\ldots,\xi_{n-1})$ and any legal sequence of q Spush, Npush, Relabel, Add edge, and ce operations following it can be executed in $O(n^3/\log n + q)$ time.*

Proof. The proof of Theorem 6.1 carries over with only two minor changes: a relabeling of a vertex v resets $z[v]$ to $\Xi_v(0)$ instead of to 0, and in the implementation of *ce* the pointer $z[v]$ steps through the blocks in the order given by Ξ_v instead of in increasing order; i.e., lines 3 and 4 of the code of *ce* are replaced by

```
    while $r'[v,z[v]] \wedge H'[h(v)-1,z[v]] = 0$ and $\Xi_v^{-1}(z[v]) < n/x-1$
    do $z[v] := \Xi_v(\Xi_v^{-1}(z[v])+1)$ od;        □
```

The incremental strongly polynomial algorithm uses the extended current-edge data type essentially as described in §7 (in the paragraph preceding Lemma 7.1). The only modification is that *generic initialize* now chooses n permutations $\xi_0,\ldots,\xi_{n-1}$ of V and calls *Init*$(\xi_0,\ldots,\xi_{n-1})$. In this situation, we say that the algorithm is executed with the adjacency lists ordered according to $\xi_0,\ldots,\xi_{n-1}$.

We now turn to the discussion of PTR events. We need the following definitions. Given finite sets A and B and permutations $\mu \in \mathrm{Perm}(A)$ and $\sigma \in \mathrm{Perm}(B)$, let $\lambda(\mu,\sigma)$, called the *coascent* of μ and σ, be the length of a longest (not necessarily contiguous) common subsequence of the sequences $\mu(0),\ldots,\mu(|A|-1)$ and $\sigma(0),\ldots,\sigma(|B|-1)$. Given l permutations $\mu_0,\ldots,\mu_{l-1}$ of subsets of a finite set A, for some $l \in \mathbb{N}$, let $\Lambda(\mu_0,\ldots,\mu_{l-1}) = \max_{\sigma\in\mathrm{Perm}(A)} \sum_{i=0}^{l-1} \lambda(\mu_i,\sigma)$; note that this quantity does not depend on A. We call $\Lambda(\mu_0,\ldots,\mu_{l-1})$ the *external coascent* of $\mu_0,\ldots,\mu_{l-1}$.

For all $u \in V$, denote by Γ_u the set of neighbors of u; i.e., $\Gamma_0 = \{v \in V : (u,v) \in E\}$. For $\xi \in \mathrm{Perm}(V)$ and $u \in V$, call $\mu \in \mathrm{Perm}(\Gamma_u)$ the *restriction* of ξ to Γ_u if the vertices in Γ_u are ordered identically by μ and by ξ, i.e., if $\mu^{-1}(v) < \mu^{-1}(w) \iff \xi^{-1}(v) < \xi^{-1}(w)$ for all v, $w \in \Gamma_u$.

LEMMA 8.3 (*see* [CH89]). *Let $\xi_0,\ldots,\xi_{n-1} \in \mathrm{Perm}(V)$. If the strongly polynomial algorithm is executed with the adjacency lists ordered according to $\xi_0,\ldots,\xi_{n-1}$, then $\sharp ptr \le 2n\cdot\Lambda(\mu_0,\ldots,\mu_{n-1})$, where μ_v is the restriction of ξ_v to Γ_v for all $v \in V$.*

Proof. For $\sigma_0,\ldots,\sigma_{2n-2} \in \mathrm{Perm}(V)$, let us say that an execution of the algorithm relabels according to $\sigma_0,\ldots,\sigma_{2n-2}$ if the following holds for $k = 0,\ldots,2n-2$ and for all v, $w \in V$: if $d(v)$ is set to $k+1$ at some point of the execution and $d(w)$ is set to $k+1$ at some later point, then $\sigma_k^{-1}(v) < \sigma_k^{-1}(w)$. Except for the fact that some vertices may not be relabeled $k+1$ times, σ_k simply orders the vertices in V by the time of their $(k+1)$st relabeling.

Consider an execution of the algorithm with the adjacency lists ordered according to $\xi_0,\ldots,\xi_{n-1}$ that relabels according to $\sigma_0,\ldots,\sigma_{2n-2}$ and fix $v \in V$ and $k \in \{0,\ldots,2n-2\}$. We will count the number $\sharp ptr_{v,k}$ of PTR events on edges with tail v while $d(v) = k$. Suppose that $w_1,\ldots,w_l$ are vertices in V such that the algorithm incurs PTR events on the edges $(v, w_1), \dots, (v, w_l)$, in that order, while $d(v) = k$. Then clearly $\mu_v^{-1}(w_1) < \dots < \mu_v^{-1}(w_l)$ and $\sigma_k^{-1}(w_1) < \dots < \sigma_k^{-1}(w_l)$; i.e., the sequences $\mu_v(1), \dots, \mu_v(|\Gamma_v|)$ and $\sigma_k(1), \dots, \sigma_k(n)$ have a (not necessarily contiguous) subsequence of length l, namely $w_1, \dots, w_l$. Thus $\sharp ptr_{v,k} \le \lambda(\mu_v, \sigma_k)$. Summing over all values of v and k yields

$$\sharp ptr \le \sum_{k=0}^{2n-2} \sum_{v \in V} \lambda(\mu_v, \sigma_k) \le \sum_{k=0}^{2n-2} \Lambda(\mu_0, \dots, \mu_{n-1}) \le 2n \cdot \Lambda(\mu_0, \dots, \mu_{n-1}). \qquad \square$$

As is clear from Lemma 8.3, our next task is to analyze $\Lambda(\mu_0, \dots, \mu_{n-1})$, where $\mu_0, \dots, \mu_{n-1}$ are obtained in various different ways.

LEMMA 8.4.

(a) *For all $v \in V$, let μ_v be a permutation of Γ_v. Then $\Lambda(\mu_0, \dots, \mu_{n-1}) \le m$.*

(b) (*See* [Al90].) *For every two integers n and h with $1 \le h \le n$ and every set W with $|W| = h$, n permutations $\mu_0, \dots, \mu_{n-1}$ of W with $\Lambda(\mu_0, \dots, \mu_{n-1}) = O(nh^{2/3})$ can be constructed in $O(nh)$ time.*

(c) *Suppose that μ_v is drawn randomly from the uniform distribution over $\mathrm{Perm}(\Gamma_v)$ for all $v \in V$ and that $\mu_0, \dots, \mu_{n-1}$ are independent. Take $\zeta = \log(2 + n(\log n)^2/m)$. Then for some $\theta = \theta(n, m)$ with $\theta = O(\sqrt{nm} + n \log n/\zeta)$ and for all $r \ge 0$,*

$$\Pr(\Lambda(\mu_0, \dots, \mu_{n-1}) \ge \theta + r) \le 2^{-r}.$$

Remark. The proof of part (c) is based on the proofs of Lemma 10 in [CH89] and of Lemma 6.3 in [CH95]. For $m = o(n(\log n)^2)$, it strengthens those lemmas.

Proof.

(a) This is obvious since $|\Gamma_0| + \dots + |\Gamma_{n-1}| = m$.

(b) This is Theorem 2 in [Al90].

(c) Recall that $\Lambda(\mu_0, \dots, \mu_{n-1}) = \max_{\sigma \in \mathrm{Perm}(V)} \phi(\sigma)$, where $\phi(\sigma) = \sum_{v=0}^{n-1} \lambda(\mu_v, \sigma)$. We will show the probability that $\phi(\sigma)$ is large to be very small for each fixed $\sigma \in \mathrm{Perm}(V)$. Multiplying that probability by the number of choices for σ, i.e., by $n!$, we obtain an upper bound on the probability that $\Lambda(\mu_0, \dots, \mu_{n-1})$ is large.

Hence let $\sigma \in \mathrm{Perm}(V)$ be arbitrary but fixed. For all $v \in V$, let $\Lambda_v = \lambda(\mu_v, \sigma)$ and take $S = \phi(\sigma) = \sum_{v=0}^{n-1} \Lambda_v$, the quantity of interest. For all $v \in V$, let d_v be the degree of v; i.e., $d_v = |\Gamma_v|$.

For arbitrary integers d and k with $0 \le k \le d \le n$, the number of permutations μ of an arbitrary subset of V of cardinality d with $\lambda(\mu, \sigma) \ge k$ is at most $\binom{d}{k}^2 (d-k)!$. To see this, note that if $\lambda(\mu, \sigma) \ge k$, then the elements of a (not necessarily contiguous) subsequence of $\mu(0), \dots, \mu(d-1)$ of length k appear in the same order in the sequence $\sigma(0), \dots, \sigma(n-1)$. The elements of the subsequence can be chosen in $\binom{d}{k}$ ways, and the positions in which they appear in $\mu(0), \dots, \mu(d-1)$ can also be chosen in $\binom{d}{k}$ ways, while the remainder of the sequence $\mu(0), \dots, \mu(d-1)$ can be chosen in $(d-k)!$ ways. It follows that for all $v \in V$ and all integers k with $1 \le k \le d_v$,

$$\Pr(\Lambda_v \ge k) \le \frac{\binom{d_v}{k}^2 (d_v - k)!}{d_v!} \le \frac{d_v^k}{(k!)^2} \le \left(\frac{e^2 d_v}{k^2} \right)^k,$$

where in the last step we used (a very crude) Stirling's approximation $k! \ge (k/e)^k$.

It can be seen that Λ_v is unlikely to exceed $\sqrt{d_v}$ by very much. Applying the Cauchy–Schwarz inequality $|u \cdot v| \le |u||v|$ to the vectors $u = (1, \dots, 1)$ and $v = (\sqrt{d_0}, \dots, \sqrt{d_{n-1}})$, we obtain

$$\sum_{v=0}^{n-1} \sqrt{d_v} \le \sqrt{n} \cdot \sqrt{\sum_{v=0}^{n-1} d_v} = \sqrt{nm}.$$

$S = \sum_{v=0}^{n-1} \Lambda_v$ is hence unlikely to exceed $\sqrt{nm}$ by very much. In order to obtain precise bounds, we use a method based on the moment-generating functions of $\Lambda_0, \dots, \Lambda_{n-1}$ and akin to the usual proof of the well-known Chernoff bounds (see, e.g., [CLR90] or [HR90]).

First, observe that for arbitrary real numbers θ, r, and t with $t \ge 1$,

$$\Pr(S \ge \theta + r) = e^{-t(\theta+r)} e^{t(\theta+r)} \Pr(e^{tS} \ge e^{t(\theta+r)}) \le e^{-t(\theta+r)} E(e^{tS}),$$

where the simple Markov inequality was used in the last step. Second, since $\mu_0, \dots, \mu_{n-1}$ and hence also $e^{t\Lambda_0}, \dots, e^{t\Lambda_{n-1}}$ are independent,

$$E(e^{tS}) = E\left(e^{\sum_{v=0}^{n-1}(t\Lambda_v)}\right) = E\left(\prod_{v=0}^{n-1} e^{t\Lambda_v}\right) = \prod_{v=0}^{n-1} E(e^{t\Lambda_v}).$$

We next bound the quantities $E(e^{t\Lambda_v})$. Let $v \in V$ and let $a_v \ge 0$ be an arbitrary integer. Then

$$\begin{aligned} E(e^{t\Lambda_v}) &= \sum_{k=0}^{\infty} e^{tk} \Pr(\Lambda_v = k) \le \sum_{k=0}^{a_v} e^{tk} \Pr(\Lambda_v = k) + \sum_{k=a_v+1}^{\infty} e^{tk} \Pr(\Lambda_v \ge k) \\ &\le e^{ta_v} \sum_{k=0}^{a_v} \Pr(\Lambda_v = k) + \sum_{k=a_v+1}^{\infty} e^{tk} \left(\frac{e^2 d_v}{k^2}\right)^k \le e^{ta_v} + \sum_{k=a_v+1}^{\infty} \left(\frac{e^{t+2} d_v}{k^2}\right)^k. \end{aligned}$$

Choose a_v to make $\frac{e^{t+2} d_v}{k^2} \le \frac{1}{2}$ for $k \ge a_v + 1$; i.e., take $a_v = \lfloor \sqrt{2e^{t+2} d_v} \rfloor$. Then

$$E(e^{t\Lambda_v}) \le e^{ta_v} + \sum_{k=a_v+1}^{\infty} 2^{-k} = e^{ta_v} + 2^{-a_v} \le 2e^{t\sqrt{2e^{t+2}d_v}} \le 2e^{te^{t+3}\sqrt{d_v}}.$$

Putting everything together yields

$$\begin{aligned} \Pr(S \ge \theta + r) &\le e^{-t(\theta+r)} E(e^{tS}) = e^{-t(\theta+r)} \prod_{v=0}^{n-1} E(e^{t\Lambda_v}) \le e^{-t(\theta+r)} \prod_{v=0}^{n-1} (2e^{te^{t+3}\sqrt{d_v}}) \\ &= e^{-t(\theta+r)} \cdot 2^n e^{te^{t+3} \sum_{v=0}^{n-1} \sqrt{d_v}} \le 2^n e^{t(e^{t+3}\sqrt{nm} - (\theta+r))}. \end{aligned}$$

Recalling that σ can be chosen in $n!$ ways, we find

$$\begin{aligned} \Pr(\Lambda(\mu_0, \dots, \mu_{n-1}) \ge \theta + r) &\le n! \cdot 2^n e^{te^{t+3}\sqrt{nm} - t\theta - tr} \\ &\le 2^{2n \log n} e^{te^{t+3}\sqrt{nm} - t\theta} e^{-tr} \le e^{2n \log n + te^{t+3}\sqrt{nm} - t\theta} \cdot 2^{-r}. \end{aligned}$$

Choose θ so as to make $2n \log n + te^{t+3}\sqrt{nm} - t\theta = 0$; i.e., take

$$\theta = \frac{2n \log n + te^{t+3}\sqrt{nm}}{t}.$$

Then $\Pr(\Lambda(\mu_0,\ldots,\mu_{n-1}) \geq \theta + r) \leq 2^{-r}$, as desired. All that remains is to show that for all combinations of n and m, it is possible to choose $t \geq 1$ such that $\theta = O(\sqrt{nm} + n\log n/\zeta)$. Consider two cases:

Case 1: $e^4\sqrt{nm} > n\log n$. In this case, take $t = 1$ and observe that $\theta = O(\sqrt{nm})$, as required. This is essentially the analysis of [CH89].

Case 2: $e^4\sqrt{nm} \leq n\log n$. Now choose $t \geq 1$ to make $te^{t+3}\sqrt{nm} = n\log n$; i.e.,

$$te^{t+3} = \sqrt{\frac{n(\log n)^2}{m}}.$$

This is clearly possible, and $t = \Omega(\zeta)$. But then $\theta = O(n\log n/\zeta)$. $\square$

THEOREM 8.1. *A maximum flow in a network with n vertices and m edges can be computed with the following bounds on flow operations and time:*

(a) *deterministically using $O(nm\log n)$ flow operations and $O(nm\log n)$ time*;

(b) *deterministically using $O(q\log n)$ flow operations and $O(nm + q\log n)$ time, where $q = n^{8/3}$.*

(c) *probabilistically using $O(\alpha q\log n)$ flow operations and $O(nm + \alpha q\log n)$ time with probability at least $1 - 2^{-\alpha\sqrt{nm}}$, for arbitrary $\alpha \geq 1$, where $q = n^{3/2}m^{1/2} + n^2\log n/\log(2 + n(\log n)^2/m)$.*

Remark. The bounds of part (a) were previously obtained by [ST83]. The time bound of part (b) was previously obtained by [Al90], although with a weaker bound on the number of flow operations. For $m = \Omega(n(\log n)^2)$, the time bound of part (c) was previously obtained by [Ta89] and [CH95], although with a weaker bound on the number of flow operations. For $m = o(n(\log n)^2)$, the result is new.

Proof.

(a) Combine Lemmas 7.5, 8.3, 8.4(a), and 6.1.

(b) Combine Lemmas 7.5, 8.3, 8.4(b) (used with $h = n$), and 8.1. (Note that with $T_{ce}(n, m, q)$ replaced by $T'_{ce}(n, m, q)$, Lemma 7.5 holds for the modified algorithm that works with the extended current-edge data type.)

(c) Initialize the current-edge data structure with n independent random permutations $\xi_0,\ldots,\xi_{n-1}$ of V. Since random permutations can be computed in linear time (see, e.g., [Se77]), this can be done in $O(n^2)$ time. Taking $r = \alpha\sqrt{nm}$ in Lemma 8.4(c) and using also Lemma 8.3, conclude that except with probability at most $2^{-\alpha\sqrt{nm}}$, we have $\sharp ptr = O(\alpha q)$. The claim now follows from Lemmas 7.5 and 8.1. $\square$

Remark. If, as in [CH89], a new random permutation ξ_v of Γ_v is computed at each relabeling of v for all $v \in V$ then the failure probability of part (c) can be reduced even further to $2^{-\alpha q}$.

Remark. Following [AOT89], we can combine the incremental wave scaling algorithm of §5 with the use of dynamic trees. Since this requires few new ideas, we omit the details and only state the following result: for every $\alpha \geq 1$, a maximum flow in a network with n vertices, m edges, and integer capacities bounded by $U \geq 1$ can be computed using $O(\alpha q\log(2 + n\log U/m) + \log U)$ flow operations and $O(nm + \alpha q\log(2 + n\log U/m) + \log U)$ time with probability at least $1 - 2^{-\alpha\sqrt{nm}}$, where $q = n^{3/2}m^{1/2} + n^2\log n/\log(2 + n(\log n)^2/m)$.

In order to use the faster solution to the extended current-edge problem provided by Lemma 8.2, we first need to demonstrate that random block-preserving permutations are almost as "good" as unrestricted random permutations. We do this by relating the external coascent of a set of block-preserving permutations to that of the set of block permutations that induces it. Recall that $x = \lfloor\log n\rfloor$.

LEMMA 8.5. *For all* $\Xi_0, \ldots, \Xi_{n-1} \in \mathrm{Perm}(M)$ *with induced block-preserving permutations* $\xi_0, \ldots, \xi_{n-1} \in \mathrm{Perm}(V)$, $\Lambda(\xi_0, \ldots, \xi_{n-1}) \le x \cdot \Lambda(\Xi_0, \ldots, \Xi_{n-1})$.

Proof. Fix $\sigma \in \mathrm{Perm}(V)$ arbitrarily and let $R \subseteq \mathrm{Perm}(M)$ be the multiset obtained as follows: for each tuple $(r_0, \ldots, r_{n/x-1}) \in B_0 \times \cdots \times B_{n/x-1}$, where r_i for $i = 0, \ldots, n/x - 1$ is called a *representative* of its block B_i, add to R (one copy of) the block permutation Ψ that arranges the blocks in the order in which their representatives occur in σ (i.e., for $0 \le i, j \le n/x - 1, \Psi^{-1}(i) < \Psi^{-1}(j) \Longleftrightarrow \sigma^{-1}(r_i) < \sigma^{-1}(r_j)$). We call $r_0, \ldots, r_{n/x-1}$ the *defining vertices* of (that copy of) Ψ. Now, for every block permutation $\Xi \in \mathrm{Perm}(M)$ with induced block-preserving permutation ξ,

$$\sum_{\Psi \in R} \lambda(\Xi, \Psi) \ge \frac{|R|}{x} \lambda(\xi, \sigma).$$

To see this, note that each element of a fixed longest common subsequence of $\xi(0), \ldots, \xi(n-1)$ and $\sigma(0), \ldots, \sigma(n-1)$ contributes 1 to $\lambda(\Xi,\ \Psi)$ if it is a defining vertex of Ψ and that each $v \in V$ is a defining vertex of exactly $|R|/x$ permutations $\Psi \in R$. Summing the inequality above for Ξ equal to $\Xi_0, \ldots, \Xi_{n-1}$ produces

$$\sum_{v=0}^{n-1} \lambda(\xi_v, \sigma) \le \frac{x}{|R|} \sum_{v=0}^{n-1} \sum_{\Psi \in R} \lambda(\Xi_v, \Psi) = \frac{x}{|R|} \sum_{\Psi \in R} \sum_{v=0}^{n-1} \lambda(\Xi_v, \Psi)$$
$$\le \frac{x}{|R|} \sum_{\Psi \in R} \Lambda(\Xi_0, \ldots, \Xi_{n-1}) = x \cdot \Lambda(\Xi_0, \ldots, \Xi_{n-1}). \qquad \square$$

We can now state the main result of our paper and finally justify its title.

THEOREM 8.2. *A maximum flow in a network with n vertices can be computed deterministically using* $O(n^{8/3}(\log n)^{4/3})$ *flow operations and* $O(n^3/\log n)$ *time.*

Proof. According to Lemma 8.4(b), used with $h = n/x$, n block permutations $\Xi_0, \ldots, \Xi_{n-1} \in \mathrm{Perm}(M)$ with $\Lambda(\Xi_0, \ldots, \Xi_{n-1}) = O(n(n/\log n)^{2/3})$ can be constructed in $O(n^2/\log n)$ time. By Lemmas 8.3 and 8.5, if the algorithm is executed with the adjacency lists ordered according to the block-preserving permutations induced by $\Xi_0, \ldots, \Xi_{n-1}$, then $\sharp ptr = O(n^{8/3}(\log n)^{1/3})$. The claim now follows from Lemmas 7.5 and 8.2. $\square$

9. Parallel algorithms. Since our solution to the current-edge problem parallelizes trivially on most parallel machines and since the current-edge problem is the only bottleneck in our algorithms on dense graphs, it is possible to crank out a variety of parallel algorithms for the maximum-flow problem that have optimal speedup, as compared with their sequential counterparts. We give one example in Theorem 9.1 below. Since we parallelize only the current-edge data structure and execute all other parts of the algorithms sequentially as before, optimal speedup can be attained only for a moderately small number of processors, as is to be expected in view of the P-completeness of the maximum-flow problem [GSS82]. Previous work on parallel algorithms for computing maximum flows is described in [SV82], [GT88], [GT89], and [Go91]. No parallel algorithm for the maximum-flow problem with optimal speedup (using more than a constant number of processors) was previously known.

THEOREM 9.1. *For* $p = O(n^{1/3}(\log n)^{-7/3})$, *a maximum flow in a network with n vertices can be computed in (optimal)* $O(n^3/(p \log n))$ *time on a network of* $2p - 1$ *processors interconnected to form a complete binary tree.*

Proof. The processor at the root of the tree stores a copy of all variables. In addition, each of the p leaf processors has a copy of the vector z, and a copy of the arrays r' and H' is distributed among the leaf processors, the ith leaf processor, for $i = 1, \ldots, p$, storing the columns of r' and H' numbered $(i-1)$, $(i-1)+p$, $(i-1)+2p$, etc. The root processor essentially carries out the algorithm of Theorem 8.2 sequentially. Each update of r' and H' is broadcast to the leaf processors and recorded by the relevant leaf processor. A call of $ce(v)$ (originating at the root) is also broadcast to the leaf processors and causes each of them to advance its copy of $z[v]$, looking only at the columns stored locally, until it encounters an admissible edge, runs out of edges, or is interrupted. A successful processor sends the admissible edge found up the tree towards the root. Whenever two edges meet in the tree, the one coming from the right is discarded. The root, upon receipt of the surviving edge, which is $ce(v)$, broadcasts it to the leaves. This signal interrupts the leaf processors and allows them to reset $z[v]$ to the correct value.

This implementation allows a sequence of q operations on the current-edge data structure to be processed in $O(q \log p + n^3/(p \log n))$ time. The algorithm of Theorem 8.2 uses $q = O(n^{8/3}(\log n)^{1/3})$ operations, giving a total time of $O(n^{8/3}(\log n)^{4/3} + n^3/(p \log n))$. For $p = O(n^{1/3}(\log n)^{-7/3})$, this is $O(n^3/(p \log n))$. $\square$

10. Open problems.

(1) Does the current-edge problem have an $o(nm)$-time solution for $m = o(n^2/\log n)$? A positive answer to this question would extend the range of $o(nm)$ algorithms below $m = \Omega(n^2/\log n)$.

(2) Can the $O(n^2 \log n / \log(2 + n(\log n)^2/m))$ term be dropped in the analysis of the number of PTR events? A positive answer to this question would extend the range of $O(nm)$ algorithms below $m = \Omega(n(\log n)^2)$.

(3) Is there an $o(nm \log n)$ maximum-flow algorithm for $m = o(n \log n / \log\log n)$?

REFERENCES

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, MA, 1974.

[AO89] R. K. Ahuja and J. B. Orlin, *A fast and simple algorithm for the maximum flow problem*, Oper. Res., 37 (1989), pp. 748–759.

[AOT89] R. K. Ahuja, J. B. Orlin, and R. E. Tarjan, *Improved time bounds for the maximum flow problem*, SIAM J. Comput., 18 (1989), pp. 939–954.

[Al90] N. Alon, *Generating pseudo-random permutations and maximum flow algorithms*, Inform. Process. Lett., 35 (1990), pp. 201–204.

[CH89] J. Cheriyan and T. Hagerup, *A randomized maximum-flow algorithm*, in Proc. 30th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1989, pp. 118–123.

[CH95] ———, *A randomized maximum-flow algorithm*, SIAM J. Comput., 24 (1995), pp. 203–226.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, and McGraw–Hill, New York, 1990.

[FF62] L. R. Ford, Jr., and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[Go91] A. V. Goldberg, *Processor-efficient implementation of a maximum flow algorithm*, Inform. Process. Lett., 38 (1991), pp. 179–185.

[GT88] A. V. Goldberg and R. E. Tarjan, *A new approach to the maximum-flow problem*, J. Assoc. Comput. Mach., 35 (1988), pp. 921–940.

[GT89] ———, *A parallel algorithm for finding a blocking flow in an acyclic network*, Inform. Process. Lett., 31 (1989), pp. 265–271.

[GSS82] L. M. Goldschlager, R. A. Shaw, and J. Staples, *The maximum flow problem is log space complete for P*, Theoret. Comput. Sci., 21 (1982), pp. 105–111.

[GLS88] M. Grötschel, L. Lovász, and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization*, Springer-Verlag, Berlin, 1988.

[HR90] T. Hagerup and C. Rüb, *A guided tour of Chernoff bounds*, Inform. Process. Lett., 33 (1990), pp. 305–308.

[Ka74] A. V. Karzanov, *Determining the maximal flow in a network by the method of preflows*, Soviet Math. Dokl., 15 (1974), pp. 434–437.

[KRT94] V. King, S. Rao, and R. Tarjan, *A faster deterministic maximum flow algorithm*, J. Algorithms, 17 (1994), pp. 447–474.

[Se77] R. Sedgewick, *Permutation generation methods*, Comput. Surveys, 9 (1977), pp. 137–164.

[SV82] Y. Shiloach and U. Vishkin, *An $O(n^2 \log n)$ parallel MAX-FLOW algorithm*, J. Algorithms, 3 (1982), pp. 128–146.

[ST83] D. D. Sleator and R. E. Tarjan, *A data structure for dynamic trees*, J. Comput. System Sci., 26 (1983), pp. 362–391.

[ST85] ———, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach., 32 (1985), pp. 652–686.

[Ta89] R. E. Tarjan, personal communication, September 1989.