SIAM J. COMPUT.
Vol. 24, No. 2, pp. 203–226, April 1995

A RANDOMIZED MAXIMUM-FLOW ALGORITHM*

JOSEPH CHERIYAN† AND TORBEN HAGERUP‡

Abstract. A randomized algorithm for computing a maximum flow is presented. For an n-vertex m-edge network, the running time is $O(nm + n^2(\log n)^2)$ with probability at least $1 - 2^{-\sqrt{nm}}$. The algorithm is always correct, and in the worst case runs in $O(nm \log n)$ time. The only use of randomization is to randomly permute the adjacency lists of the network vertices at the start of the execution.

The analysis introduces the notion of premature target relabeling (PTR) events and shows that each PTR event contributes $O(\log n)$ amortized time to the overall running time. The number of PTR events is always $O(nm)$; however, it is shown that when the adjacency lists are randomly permuted, then this quantity is $O(n^{3/2}m^{1/2} + n^2 \log n)$ with high probability.

Key words. maximum flow, randomized algorithm, random permutations, scaling, dynamic tree, Fibonacci heap

AMS subject classifications. 68Q20, 68Q25, 68R05, 90C35

1. Introduction.

A *network* is a graph together with one or more functions from the edges to the real numbers. Problems on networks arise often in theory and in practice. One of the central problems in this area is that of finding a *maximum flow* in a network. The input to the problem consists of a graph with two distinguished vertices, the *source* and the *sink*, and a nonnegative function on the edges called the *capacity*. Let n and m denote the number of vertices and edges, respectively, and let U denote the maximum capacity of any edge.

The first algorithm for the problem was presented by Ford and Fulkerson [FF57], [FF62]. Their algorithm does not run in polynomial time; moreover, when the capacities are irrational, it may not even terminate. More than a decade later, Edmonds and Karp [EK72] proposed a modification of the algorithm of [FF57] that runs in polynomial time. Independently, Dinic [D70] gave a faster polynomial-time algorithm.

The running times of the Dinic and Edmonds–Karp algorithms can be bounded by functions that depend only on n and m, but not on the actual edge capacities. This behavior is considered to be attractive and has been studied in the more general setting of combinatorial optimization. The input to a combinatorial optimization problem consists of a discrete structure (e.g., the graph in the maximum-flow problem), together with numeric parameters (e.g., the edge capacities in the maximum-flow problem). For an instance of the problem, let p denote the size of (a reasonable encoding of) the discrete structure, and let ℓ denote the maximum size of any numeric parameter (e.g., if the numeric parameters are integers with absolute values bounded by U, then one can take $\ell = 1 + \lceil \log(U+1) \rceil$; see Grötschel, Lovász, and Schrijver [GLS88]). An algorithm is said to be *strongly polynomial* if its running time in the arithmetic model (i.e., the total number of arithmetic operations, comparisons, and data transfers executed) is polynomial in p, and the maximum size of any number computed by the algorithm is polynomial in $p\ell$. The maximum-flow algorithms in [D70] and [EK72] are strongly polynomial.

*Received by the editors November 1, 1991; accepted for publication (in revised form) October 21, 1993. A preliminary and abridged version of this paper was presented at the 30th IEEE Symposium on Foundations of Computer Science in October 1989. Most of the research was carried out while both authors were at the Fachbereich Informatik of the Universität des Saarlandes, Saarbrücken, Germany.

†Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada. This research was supported in part by the ESPRIT II Basic Research Actions Program of the European Community under contract no. 3075 (project ALCOM) and by the National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research, through National Science Foundation grant DMS-8920550.

‡Max-Planck-Institut für Informatik, D-66123 Saarbrücken, Germany. This research was supported in part by the ESPRIT II Basic Research Actions Program of the European Community under contract no. 3075 (project ALCOM).

The publication of [D70] and [EK72] led to further research on maximum-flow algorithms. Karzanov [K74] introduced the notion of a *preflow*, a generalization of a flow, and gave an algorithm that computes a maximum flow in $O(n^3)$ time by manipulating a preflow. Other improvements in the time complexity followed, and this line of research culminated in [ST83], where Sleator and Tarjan introduced a new data structure for dynamic trees and showed that by using this data structure, a maximum flow can be computed in $O(nm \log n)$ time. For graphs with $O(n^2/(\log n)^3)$ edges, this running time was the fastest then known. For networks with integer capacities, Gabow [Ga85] subsequently reported a simple algorithm based on the scaling technique with a running time of $O(nm \log U)$; under the so-called *similarity assumption*, i.e., $U = n^{O(1)}$, this algorithm is as fast as the one in [ST83].

A new approach for computing maximum flows was presented by Goldberg and Tarjan [GT88], based on earlier work by Goldberg [G85]. [GT88] introduced a generic algorithm for computing a maximum flow that works by manipulating a preflow, and gave a specific algorithm that uses the dynamic trees data structure and runs in time $O(nm \log(n^2/m))$. This improves on the algorithm in [ST83] for relatively dense graphs with $m = n^{2-o(1)}$. Cheriyan and Maheshwari [CM89] investigated several specific instances of the generic algorithm and showed that one of them runs in $\Theta(n^2\sqrt{m})$ time. For the special case of integer capacities, Ahuja and Orlin [AO89] devised an algorithm based on the scaling technique, called the *excess scaling algorithm*, that runs in $O(nm + n^2 \log U)$ time. Ahuja, Orlin, and Tarjan [AOT89] obtained several fast algorithms for this case by combining the dynamic trees data structure with the excess scaling algorithm.

At a somewhat general level, several aspects of the present paper were influenced by recent research on the *minimum-cost flow problem*. Edmonds and Karp [EK72], introducing the scaling technique, gave a polynomial minimum-cost flow algorithm for the special case of integer capacities. The algorithm, however, is not strongly polynomial. Edmonds and Karp left open the challenging question of whether a strongly polynomial algorithm exists. Although this question attracted research interest, no significant progress was reported for over a decade, until Tardos [T85] answered the question in the affirmative. There followed a number of other strongly polynomial algorithms: Fujishige [F86], Galil and Tardos [GaT88], Goldberg and Tarjan [GT89], [GT90], and Orlin [O88]. An idea common to these papers was the use of the scaling method for designing efficient strongly polynomial algorithms, and the most efficient realization of this idea was given in [O88], where the Edmonds–Karp capacity-scaling algorithm was "adapted" to real-valued capacities in order to achieve the fastest strongly polynomial running time known. The design of our algorithm was also motivated by this idea.

The focus of our work is on using randomization to improve on the Sleator–Tarjan time bound of $O(nm \log n)$ for finding a maximum flow. We show that by suitably incorporating the dynamic trees data structure into the excess scaling algorithm *and* randomly permuting the adjacency lists of the network vertices at the start of the execution, we can compute a maximum flow in $O(nm + n^2(\log n)^2)$ time with high probability, and in $O(nm \log n)$ time in the worst case.

The running time analysis introduces the notion of *premature target relabeling* (PTR) events and shows that each PTR event contributes $O(\log n)$ amortized time to the overall running time. The number of PTR events is always $O(nm)$; however, it is shown that when the adjacency lists are randomly permuted, then this quantity is $O(n^{3/2}m^{1/2} + n^2 \log n)$ with high probability.

Section 3 discusses a variant of the excess scaling algorithm, and in §4 an efficient strongly polynomial algorithm is designed, based on the algorithm in §3. The running time is analyzed in §§5–8. The critical component of the running time is identified in §5, and it is shown that the algorithm is strongly polynomial. Section 6 introduces PTR events and gives a probabilistic

bound on the number of these events, assuming that the adjacency lists are permuted randomly. The analysis is completed in §§7 and 8, first using simple arguments that lead to a somewhat loose time bound, and then using a more refined argument that gives the tightest bound known on the running time. Some conclusions are presented in §9.

2. Preliminaries. We use the traditional model for the study of problems on networks. Capacities and flow values are represented by real numbers, and all other quantities are represented by integers. For n-vertex input networks, we allow integers of absolute value $n^{O(1)}$, and we charge constant time for each arithmetic operation (i.e., addition, subtraction, comparison, or division by 2) on real numbers or integers, and for each data transfer. Further, we charge constant time for generating one random integer.

A *flow network* consists of a directed graph $G = (V, E)$, assumed to be symmetric (i.e., $(v, w) \in E \iff (w, v) \in E$), and a function $cap : E \to \mathbb{R}_+ \cup \{0\}$, the *capacity*, together with two distinguished vertices, the *source*, s, and the *sink*, t. Let $n = |V|$ and $m = |E|$, and let U denote the maximum capacity of any edge. We assume that $m \geq n \geq 3$ and that $U > 0$. For $v \in V$, let $\Gamma(v) = \{w \in V : (v, w) \in E\} = \{u \in V : (u, v) \in E\}$, and for every $(v, w) \in V \times V$, let $tail(v, w) = v$ and $head(v, w) = w$.

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

(1) $f(w, v) = -f(v, w)$ for all $(v, w) \in E$ (antisymmetry constraint);

(2) $f(v, w) \leq cap(v, w)$ for all $(v, w) \in E$ (capacity constraint);

(3) $\sum_{u \in \Gamma(v)} f(u, v) \geq 0$ for all $v \in V - \{s\}$ (nonnegativity constraint).

A preflow f in G is a *flow* if $\sum_{u \in \Gamma(v)} f(u, v) = 0$ for all $v \in V - \{s, t\}$ (flow conservation constraint). The *value* of f is $\sum_{u \in \Gamma(t)} f(u, t)$, and a *maximum flow* in G is a flow in G of maximum value.

For a fixed preflow f under consideration and for every vertex $v \in V$, the *flow excess* of v, $e(v)$, is defined as $\sum_{u \in \Gamma(v)} f(u, v)$, i.e., the net flow into v. A vertex v is called *active* if $v \in V - \{s, t\}$ and $e(v) > 0$.

An edge $(v, w) \in E$ is *residual* (with respect to f) if $f(v, w) < cap(v, w)$. The *residual capacity* of an edge (v, w), $rescap(v, w)$, is defined to be $cap(v, w) - f(v, w)$.

A *labeling* of G is a function $d : V \to \mathbb{N}_0 = \{0, 1, 2, \ldots\}$. The labeling is *valid* for G and a preflow f in G exactly if $d(t) = 0$, $d(s) = n$, and $d(v) \leq d(w) + 1$ for each residual edge (v, w). We call a residual edge (v, w) *eligible* if $d(v) = d(w) + 1$.

We now describe a generic algorithm that uses a preflow f in G and a labeling d of G. The basic procedure for manipulating f is as follows:

procedure *push*$((v, w) : edge; c : real)$;
    precondition: v is active, (v, w) is eligible and $0 < c \leq \min\{e(v), rescap(v, w)\}$.
        $f(v, w) := f(v, w) + c$; $f(w, v) := f(w, v) - c$;
        $e(v) := e(v) - c$; $e(w) := e(w) + c$;

By a *push* over (v, w) of *value* c we mean an execution of *push* with parameters (v, w) and c. v and w are called the *tail vertex* and the *head vertex* of the push, respectively. The push is called *saturating* if $f(v, w) = cap(v, w)$ afterwards; otherwise it is called *nonsaturating*. For a vertex v, a push *out of* v is a push over any edge of the form (v, w), and a push *into* v is a push over any edge of the form (u, v).

The basic procedure for manipulating the labeling d is as follows:

procedure *relabel*$(v : vertex)$;
    precondition: v is active, and no edge (v, w) emanating from v is eligible.
        $d(v) := d(v) + 1$;

An execution of *relabel*(v) is called a *relabeling* of v.

The generic algorithm starts by setting the flow on each edge that leaves the source equal to its capacity and the flow on each edge that is not incident with the source equal to zero. It also fixes some initial valid labeling. The algorithm then repeatedly executes *push* and *relabel* operations in any order. When no *push* or *relabel* operation has its precondition satisfied, i.e., when there are no more active vertices, the algorithm terminates. A formal description follows.

procedure *generic initialize*;
  for all $(v, w) \in E$ **do** $f(v, w) := 0$;
  for all $v \in V$ **do** $e(v) := 0$;
  for all $(s, v) \in E$ **do**
  begin
    $f(s, v) := cap(s, v)$;
    $f(v, s) := -f(s, v)$;
    $e(v) := cap(s, v)$;
  end;
  for all $v \in V - \{s\}$ **do** $d(v) := 0$;
  $d(s) := n$;

The generic maximum-flow algorithm:
  generic initialize;
  while there is an active vertex **do**
    execute some *push* or *relabel* operation whose precondition is satisfied.

There are some minor differences between the algorithm above and the generic maximum-flow algorithm of Goldberg and Tarjan [GT88]: the *push*(v, w) operation of [GT88] always sends $c = \min\{e(v), rescap(v, w)\}$ units of flow, and the *relabel*(v) operation of [GT88] sets $d(v)$ to $\min\{d(w) + 1 : w \in \Gamma(v)$ and $rescap(v, w) > 0\}$. Despite these differences, our algorithm shares the essential properties of the generic algorithm of Goldberg and Tarjan. In particular, the proofs given in [GT88] of the next two lemmas and of Theorem 2.4 carry over without modification.

LEMMA 2.1 [GT88]. *For all* $v \in V, 0 \leq d(v) \leq 2n - 1$ *throughout the execution. In particular, the total number of relabeling operations is* $\leq n(2n - 1) \leq 2n^2$.

LEMMA 2.2 [GT88]. *The total number of saturating pushes is* $O(nm)$.

Our analysis uses the notion of an *undirected edge* $\{v, w\}$, i.e., a pair of directed edges (v, w) and (w, v). Define the capacity of an undirected edge $\{v, w\}$ to be $ucap(v, w) = cap(v, w) + cap(w, v)$. For all $v \in V$, let $\deg(v)$ denote the number of undirected edges incident with v.

Each vertex $v \in V$ has an *adjacency list*, which consists of all edges $(v, w) \in E$. For each $v \in V$, the first eligible edge (if any) in v's adjacency list is called its *current edge* and is denoted by $ce(v)$ ($ce(v) = nil$, if there are no eligible edges (v, w)). We study implementations of the generic maximum-flow algorithm that maintain, for each vertex v, a pointer to $ce(v)$. Lemma 2.3, whose proof is similar to those of Lemma 4.1 and Theorem 4.2 in [GT88], shows that this contributes $O(nm)$ time to the total running time.

LEMMA 2.3. *Maintaining current edges for all vertices over the whole execution can be done in* $O(nm)$ *time.*

Proof. Maintain for each vertex $v \in V$ a pointer z_v into the adjacency list of v. At each relabeling of v, initialize z_v to point to $ce(v)$. Whenever the edge pointed to by z_v becomes ineligible, step z_v through v's adjacency list, starting from its previous position, until it either points to an eligible edge or reaches the end of the list. The fact that z_v always points to $ce(v)$, or to the end of v's adjacency list if $ce(v) = nil$, follows from the observation that once an

edge (v, w) becomes ineligible, it remains so until the next relabeling of v. For each relabeling of v, the total time spent in maintaining z_v is $O(\deg(v))$. Summing over all relabelings of all vertices gives a total time of $O(\sum_{v \in V}(2n-1)\deg(v)) = O(nm)$. □

THEOREM 2.4 [GT88]. *Suppose that the algorithm terminates. Then, at termination, the preflow f is a maximum flow.*

3. The excess scaling algorithm. In this section we consider networks with integer capacities. We give a variant of the excess scaling algorithm of Ahuja and Orlin [AO89] and show that although this algorithm is not strongly polynomial, the number of pushes executed by it is $O(n^2 m)$. In the next section, we make appropriate modifications to the algorithm to obtain a strongly polynomial algorithm.

The excess scaling algorithm that we are about to present is an instance of the generic algorithm of §2, and except for the minor differences noted there, it is also an instance of the generic maximum-flow algorithm of Goldberg and Tarjan [GT88]. The implementation of the generic algorithm in [GT88, p. 929] repeatedly selects an active vertex and applies a *push/relabel* step to it. Rather than selecting an arbitrary active vertex, it turns out to be advantageous to select an active vertex with relatively large flow excess. The excess scaling algorithm maintains a parameter Δ; initially $\Delta = 2^{\lfloor \log U \rfloor}$, and $\Delta = 0$ at termination. The execution of the algorithm is partitioned into *phases* such that Δ stays fixed during each phase and decreases by a factor of 2 between consecutive phases. We use "Δ" to denote both this parameter and its value at some step of the execution; the context will resolve any ambiguity. By a Δ-*phase*, where $\Delta \in \mathbb{R}$, we mean a phase in which the value of the parameter equals Δ.

Consider a particular phase. At the start of the phase the algorithm satisfies the invariant that $e(v) \leq 2\Delta$, for all $v \in V - \{s, t\}$. The algorithm selects an active vertex v only if $e(v) \geq \Delta$. Furthermore, among all such vertices it selects one with minimum $d(v)$. Another invariant for the Δ-phase is that every push has value $\leq 2\Delta$. It follows that for all $v \in V - \{s, t\}$, $e(v) < 3\Delta$ always (cf. Fact 3.4).

To satisfy the 2Δ bound on the value of every push, the value of every push out of a vertex $v \in V - \{s, t\}$ is computed using a modification $\tilde{e}(v)$ of $e(v)$ defined as follows:

$$\text{If } e(v) \leq 2\Delta, \text{ then } \tilde{e}(v) = e(v), \text{ otherwise } \tilde{e}(v) = \Delta.$$

This achieves two things: First, the amount of flow sent is always $\leq 2\Delta$, as claimed above. Second, if at some point $e(v) > 2\Delta$, then the algorithm attempts to make $e(v)$ equal to zero during the same phase by selecting v twice. (This would not be achieved, for example, by letting $\tilde{e}(v) = \min\{e(v), 2\Delta\}$.)

```
procedure ẽ(v : vertex) : real;
    return (if e(v) ≤ 2Δ then e(v) else Δ);
procedure select: vertex;
  precondition: ∃v ∈ V − {s, t} : e(v) ≥ Δ.
    Return any vertex v ∈ V − {s, t} with e(v) ≥ Δ and
      d(v) = min{d(u) : u ∈ V − {s, t} and e(u) ≥ Δ};

The excess scaling algorithm:
    generic initialize;
    Δ := 2^⌊log U⌋;
    while Δ ≥ 1 do
    begin
      while ∃v ∈ V − {s, t} : e(v) ≥ Δ do
      begin
        v := select;
```

```
        while ce(v) ≠ nil and rescap(ce(v)) ≤ ẽ(v) do
            push (ce(v), rescap(ce(v)));    (* a saturating push *)
        if ce(v) = nil and ẽ(v) > 0 then
            relabel(v)
        else
            if e(v) ≥ Δ then
                push (ce(v), ẽ(v));         (* a nonsaturating push *)
    end;
    Δ := ⌊Δ/2⌋;
end.
```

To prove the correctness of the algorithm and to analyze its running time, we need the following facts. These facts are also used in subsequent sections.

Fact 3.1. For every vertex $w \in V - \{s, t\}$, $e(w)$ does not increase while $e(w) \geq \Delta$.

Proof. Any vertex v selected while $e(w) \geq \Delta$ has $d(v) \leq d(w)$, by definition. Since pushes are executed out of selected vertices only, it follows that the head vertex u of each push has $d(u) < d(w)$. In other words, no flow is sent into w while $e(w) \geq \Delta$. □

Fact 3.2. The value of every push is $\leq 2\Delta$.

Proof. This follows from the definition of $\tilde{e}$. □

Fact 3.3. The value of every nonsaturating push is $\geq \Delta$.

Fact 3.4. For every $v \in V - \{s, t\}$, $e(v) \leq 2\Delta$ at the start of each phase, $e(v) < \Delta$ at the end of each phase, and $e(v) < 3\Delta$ always.

Proof. Consider any $v \in V - \{s, t\}$. After initialization, $e(v) \leq 2\Delta$ clearly holds. At the termination of each phase, $e(v) < \Delta$, and hence $e(v) \leq 2\Delta$ at the start of the next phase. Facts 3.1 and 3.2 together imply that $e(v) < 3\Delta$ always. □

THEOREM 3.5 [AO89]. *The excess scaling algorithm is partially correct.*

Proof. The following claim shows that the termination condition for the generic algorithm is satisfied; the correctness then follows from Theorem 2.4.

Claim. When the last phase (with $\Delta = 1$) terminates, there are no active vertices.

To see this, use induction on the number of steps executed to show that the preflow f, and hence also the flow excess e, has integer values throughout. Therefore, when the last phase terminates, $e(v) = 0$ for all $v \in V - \{s, t\}$ (Fact 3.4). □

The next two results are not used later; however, they are of interest since the first one shows that the number of push steps can be bounded independently of U, while the second one uses this to give a new bound on the running time.

LEMMA 3.6. *The algorithm executes* $O(n^2 \min\{m, \log U\})$ *nonsaturating pushes.*

Proof. Ahuja and Orlin [AO89] gave an $O(n^2 \log U)$ bound on the number of nonsaturating pushes for their algorithm. Since the algorithmm here is a variant of the one in [AO89], a variant of their analysis applies here and gives the same $O(n^2 \log U)$ bound. We show the bound of $O(n^2 m)$ using a potential argument. First, for each $v \in V$, let

$$\phi(v) = \begin{cases} 0, & \text{if } e(v) = 0, \\ 1, & \text{if } 0 < e(v) \leq 2\Delta, \\ 2, & \text{if } e(v) > 2\Delta. \end{cases}$$

Intuitively, $\phi(v)$ is the number of times that v can be selected before $e(v)$ drops to zero (cf. the definition of $\tilde{e}$). Now take

$$\Phi = \sum_{v \in V - \{s,t\}} \phi(v) \cdot d(v).$$

A nonsaturating push over an edge (v, w) always lowers $\phi(v)$ by 1 and never increases $\phi(w)$ by more than 1 (Fact 3.2). Since $d(v) = d(w) + 1$, it follows that every nonsaturating push decreases Φ by at least 1. By Lemma 2.1, a saturating push increases Φ by at most $2n$, and a relabeling clearly increases Φ by at most 2. Furthermore, only saturating pushes and relabelings can increase Φ; in particular, note that going from one phase to the next leaves Φ unchanged. $\Phi = 0$ initially and $\Phi \geq 0$ always, so the total decrease in Φ (due to nonsaturating pushes) is bounded by the total increase in Φ (due to saturating pushes and relabelings). Since there are $O(nm)$ saturating pushes (Lemma 2.2) and $O(n^2)$ relabelings (Lemma 2.1), the number of nonsaturating pushes is $O(n \cdot nm + n^2) = O(n^2 m)$. □

THEOREM 3.7. *The excess scaling algorithm runs in time* $O(nm + n \log U + n^2 \min\{m, \log U\})$.

Proof. The initialization can be done in time $O(m)$. Each *relabel* or *push* step takes time $O(1)$. When the *select* procedure is implemented using appropriate data structures [AO89], [GTT90], it contributes an overall running time proportional to the number of pushes plus $O(n)$ per phase. The number of phases is $\lfloor \log U \rfloor + 1$. The theorem now follows from Lemmas 2.1–2.3 and 3.6. □

The next fact is very useful and has a straightforward proof. In the context of the minimum-cost flow problem, analogous results are crucial for designing and analyzing efficient strongly polynomial algorithms (cf. Lemma 4 of [F86] and Lemma 6 of [O88]). The proof technique used here resembles that of Lemma 6 in [O88].

Fact 3.8. For every fixed Δ-phase and every $(v, w) \in E$, the increase of $f(v, w)$ during the phase and all subsequent phases is at most $20n^2\Delta$.

Proof. Focus first on the Δ-phase and consider the potential

$$\Phi = \sum_{u \in V - \{s,t\}} e(u) \cdot d(u).$$

Whenever $f(v, w)$ increases by c (due to a push over (v, w) with value c), Φ decreases by at least c. The total increase of $f(v, w)$ during the phase is therefore bounded by the total decrease of Φ. $\Phi \geq 0$ always, and at the start of the phase $\Phi \leq 4n^2\Delta$ (by Lemma 2.1 and Fact 3.4). No *push* operation increases Φ, and a *relabel* operation increases Φ by at most 3Δ (by Fact 3.4). Consequently, the total increase of Φ is at most $6n^2\Delta$ (by Lemma 2.1). It follows that the total decrease of Φ is at most $10n^2\Delta$. Therefore, the total increase of $f(v, w)$ during the phase is at most $10n^2\Delta$.

The result now follows by summing over the remaining phases and using the fact that Δ decreases by a factor of 2 between consecutive phases. □

We call (v, w) a *forward huge edge* if $rescap(v, w) > 20n^2\Delta$. By Fact 3.8, a forward huge edge will never again be saturated. An undirected edge $\{v, w\}$ is called *huge* if $ucap(v, w) > 40n^2\Delta$. If $\{v, w\}$ is huge, clearly one of the directed edges (v, w) or (w, v), say, (v, w), is a forward huge edge. If the reverse edge (w, v) is not also a forward huge edge, we call it a *reverse huge edge*. Notice that an (undirected) huge edge continues to be huge until the end of the execution. We denote the total number of saturating pushes over (reverse) huge edges by $\sharp$*huge sat pushes*.

4. The strongly polynomial algorithm. The excess scaling algorithm has two bottlenecks that make it inefficient in comparison with previously known fast strongly polynomial algorithms. The first bottleneck is the large amount of time spent in executing nonsaturating pushes. The standard way of avoiding this bottleneck is to store for each vertex v the current edge $ce(v)$ together with $rescap(ce(v))$ in a suitable data structure that supports the operation of sending flow over a path of stored current edges. Such data structures make it possible to send flow over a path of l current edges in a running time that is substantially less than

proportional to l. The most efficient data structure known for this purpose is the *dynamic trees* data structure of Sleator and Tarjan [ST83].

From the point of view of achieving a running time of $O(nm)$, this data structure has one drawback. When used in the straightforward way, it contributes a running time of $O(\log n)$ per saturating push, i.e., of $O(nm \log n)$ overall, since the number of saturating pushes is $O(nm)$. In the context of the generic algorithm of Goldberg and Tarjan, a more efficient way of using the data structure is to insert an edge only if at least one nonsaturating push will be executed over the edge. In other words, the current edge of a vertex v is inserted into the data structure only when a push is to be applied to v while $rescap(ce(v)) > \tilde{e}(v)$, i.e., if the push operation will not saturate v's current edge.

We do not know whether this simple heuristic alone decreases the number of dynamic trees operations sufficiently. However, we obtain a remarkable decrease by combining it with another heuristic:

> At the start of the execution, randomly permute
> the adjacency list of each vertex.

The effectiveness of this heuristic comes from its interaction with scaling. As the execution progresses and the parameter Δ decreases, many edges eventually become huge. Roughly speaking, in a situation where there are many forward huge edges, the sum over all vertices v of the number of times v changes its current edge is significantly less than nm, unless the adjacency lists are maliciously ordered. A quantitative analysis of the efficiency of randomly permuting the adjacency lists needs the notion of PTR events, and we present such an analysis in §6 after developing the necessary machinery.

The second bottleneck of the excess scaling algorithm is that there are $\lfloor \log U \rfloor + 1$ phases, each of which incurs a running time overhead of $O(n)$ for initializing data structures. This bottleneck is easy to avoid. Rather than using the standard scaling method of decreasing the parameter Δ by a factor of 2 between phases, we use "tight scaling" and decrease Δ as much as possible between phases, i.e., at the end of each phase Δ is set to the minimum of $\Delta/2$ and $\max\{e(v) : v \in V - \{s, t\}\}$. We use a heap data structure to store all the vertices $v \in V - \{s, t\}$, with the key $e(v)$ used to maintain the heap order. Tight scaling with a running time overhead of $O(\log n)$ per phase is easy to implement in this way.

A *heap* is a data structure that maintains a set of *items*, each with a real-valued *key*, under the following operations [FT87]:

make heap:	Return a new empty heap.
is empty(h):	If the heap h has no items, then return *true*; otherwise return *false*.
insert(i, h):	Insert a new item i with predefined key into the heap h.
find max(h):	Return an item of maximum key in the heap h.
delete max(h):	Delete an item of maximum key from the heap h and return it.
delete(i, h):	Delete the item i from the heap h.
increase key(c, i, h):	Increase the key of the item i in the heap h by adding the nonnegative real number c.

The two last operations assume that the position of i in h is known. The *Fibonacci heaps* data structure [FT87] supports a sequence of k *delete* or *delete max* operations and l other heap operations, starting with no heaps, in time $O(l + k \log l)$.

The actual implementation of the strongly polynomial algorithm uses two heaps, rather than one, in order to execute the *select* step efficiently. The *d_heap* contains all vertices $v \in V - \{s, t\}$ with $e(v) \geq \Delta$. The heap order is maintained according to the key $-d(v)$. The *d_heap* is needed for efficiently selecting a vertex v with minimum $d(v)$ among those

with $e(v) \geq \Delta$. The *e_heap* is a Fibonacci heap containing all vertices $v \in V - \{s, t\}$ with $e(v) < \Delta$. The heap order is maintained according to the key $e(v)$. The *e_heap* is needed for efficient updating of Δ. A Fibonacci heap is used because only $O(1)$ amortized time can be allowed per *increase key* operation.

The *dynamic trees* data structure of Sleator and Tarjan [ST83], [ST85], [T83] maintains a set of vertex-disjoint rooted trees in which each tree edge has an associated real value and is considered to be directed toward the root, i.e., from child to parent. We shall need the following dynamic trees operations:

find root(v):	Find and return the root of the tree containing the vertex v.
find value(v):	Find and return the value of the edge from the vertex v to its parent. This operation assumes that v is not a tree root.
find min(v):	Return the nonroot ancestor w of v with minimum *find value*(w). In the case of ties, the ancestor furthest from v is returned. This operation assumes that v is not a tree root.
add value(v, c):	Add the real number c to the value of every edge on the path from the vertex v to *find root*(v).
link(v, w, c):	Combine the trees containing the vertices v and w by making w the parent of v and giving the value c to the new edge from v to w. This operation assumes that v and w are in different trees and that v is a tree root.
cut(v):	Break the tree containing the vertex v into two trees by deleting the edge from v to its parent. This operation assumes that v is not a tree root.

A sequence of k dynamic trees operations, starting with a collection of n single-vertex trees, can be executed in $O(k \log n)$ time.

We use the dynamic trees data structure to maintain a spanning forest F of G that contains a subset of the current edges, where the value associated with an edge in F is its residual capacity. The algorithm represents the preflow f in one of two different ways: For $(v, w) \in E$, if $(v, w) \notin F$ and $(w, v) \notin F$, then $f(v, w)$ is stored explicitly (using an array $f : E \to \mathbb{R}$). Otherwise, if $(v, w) \in F$, then $f(v, w)$ is given implicitly by $cap(v, w) - rescap(v, w)$ and $f(w, v)$ is given by $-f(v, w)$. Whenever a tree edge (v, w) is cut, we must find its associated value (i.e., $rescap(v, w)$) and then update the current values of $f(v, w)$ and $f(w, v)$. Also, when the algorithm terminates, $f(v, w)$ and $f(w, v)$ must be computed for all $(v, w) \in F$. The procedures *Link* and *Cut* given below execute a *link* and a *cut* and incorporate these conventions for representing f.

procedure *Link*$(v : vertex)$;
  (* Insert the edge $ce(v)$ into F *)
    Let $w = head(ce(v))$;
    link$(v, w,$ *rescap*$(v, w))$;

procedure *Cut*$(v : vertex)$;
  (* Cut the tree edge $ce(v) \in F$ and restore f values *)
    Let $w =$ *head*$(ce(v))$;
    $f(v, w) := cap(v, w) -$ *find value*(v); $f(w, v) := -f(v, w)$;
    cut(v).

The heart of the algorithm is the procedure *macropush*. The algorithm repeatedly selects a vertex $v \in V - \{s, t\}$ with $e(v) \geq \Delta$ that has minimum label $d(v)$ among the vertices with flow excess $\geq \Delta$, after which the *macropush* procedure is applied to v.

If v is a nonroot vertex in F, i.e., $v \neq$ *find root*(v), then *macropush* uses the dynamic trees data structure to send flow from v, over a path of current edges, to *find root*(v). If one or

more edges become saturated, then all saturated edges in F are deleted using *cut* operations. The algorithm may execute more than one nonsaturating push while sending flow over a path in F.

Now suppose that v is a root in F. Then *macropush* executes zero or more saturating pushes over edges emanating from v, without inserting these edges into the dynamic trees data structure. After this, provided that the remaining flow excess of v is at least $\Delta/2$ and that there is an eligible edge emanating from v, *macropush* executes a sequence consisting of a *Link*(v) operation and a nonsaturating push over $ce(v)$. The reason for executing a *Link*(v) only if $e(v) \geq \Delta/2$ is to ensure that only edges with sufficiently large capacities relative to Δ are ever inserted into the dynamic trees data structure (cf. Fact 5.2).

In the following outline of the algorithm, the operations on heaps are not mentioned explicitly. The procedure $\tilde{e}(v)$ is repeated from the previous section, and the procedures *relabel* and *select* are more elaborate versions of identically named procedures in previous sections.

```
procedure  ẽ(v : vertex) : real;
    return (if e(v) ≤ 2Δ then e(v) else Δ);

procedure  relabel(v : vertex);
    for all u ∈ V with ce(u) = (u, v) do
        if (u, v) ∈ F then Cut(u);
    d(v) := d(v) + 1;

procedure  select: vertex;
    (* Return an active vertex v with e(v) ≥ Δ and minimum d(v) among the vertices having
    flow excess ≥ Δ, or decrease Δ and then select v as before, or return with Δ = 0 *)
    if ∀v ∈ V − {s, t} : e(v) < Δ then
        Δ := min{Δ/2, max{e(v) : v ∈ V − {s, t}}};
    if Δ > 0 then
        let v be any active vertex with e(v) ≥ Δ and
            d(v) = min{d(u) : u ∈ V − {s, t} and e(u) ≥ Δ}
    else
        let v := t;    (* dummy value *)
    return (v);

procedure  macropush(v : vertex);
    (* Send flow from v until v is relabeled or until e(v) decreases to < Δ/2 due to saturating
    pushes out of v or until flow is sent from v over a path in F *)
    if v = find root(v) then
        begin
        while ce(v) ≠ nil and rescap(ce(v)) ≤ ẽ(v) do
            push(ce(v), rescap(ce(v)));    (* a saturating push *)
        if ce(v) = nil and ẽ(v) > 0 then
            begin relabel(v); return; end
        else
            if ẽ(v) < Δ/2 then return
            else
                Link(v);    (* insert ce(v) with rescap(ce(v)) > ẽ(v) ≥ Δ/2 *)
        end;
```

(* Send as much flow as possible from v to *find root*(v), and then cut the tree edges in F that get saturated *)
$c := \min\{$*find value*(*find min*$(v))$, $\tilde{e}(v)\}$;
change value$(v, -c)$;
$e(v) := e(v) - c$; e(*find root*$(v)) := e$(*find root*$(v)) + c$;
while $v \neq$ *find root*(v) and *find value*(*find min*$(v)) = 0$ **do** *Cut*(*find min*$(v))$;

The strongly polynomial algorithm:

generic initialize;
initialize the *d_heap*, the *e_heap* and the dynamic trees data structure;
for each $v \in V$, randomly permute the adjacency list of v;
$\Delta := \infty$;
loop
  $v :=$ *select*;
  if $\Delta = 0$ **then stop;** (* f is a maximum flow *)
  macropush(v);
forever.

THEOREM 4.1. *The strongly polynomial algorithm is partially correct.*

5. Preliminaries for the analysis. After presenting some elementary facts about the strongly polynomial algorithm, we give a lemma that identifies the critical component of its running time.

The algorithm is easily seen to satisfy Facts 3.1, 3.2, 3.4, and 3.8.

Fact 5.1. Between consecutive phases, Δ decreases by a factor of 2 or more.

Fact 5.2. When a *link* operation is executed on an edge (v, w), then *rescap*$(v, w) \geq \Delta/2$.

Fact 5.3. For all $v \in V$, any increase of $e(v)$ while $v \neq$ *find root*(v) is caused by saturating pushes into v.

Proof. Any flow sent into v by a nonsaturating push is sent further all the way to *find root*(v); hence a nonsaturating push causes $e(v)$ to increase only if $v =$ *find root*(v). □

Fact 5.4. For all $v \in V$, if an execution of *macropush*(v) starting with $v \neq$ *find root*(v) does not execute any *cut*, then either $e(v) = 0$ or $e(v) > \Delta$ when the procedure terminates.

Proof. Clearly the total amount of flow sent from v by the procedure is $c = \tilde{e}(v)$. The claim now follows from the definition of $\tilde{e}(v)$. □

By a *select step* we mean one iteration of the main loop (**loop** . . . **forever**) of the algorithm. A vertex v is said to be *processed* by a *select* step if the call of the *select* procedure in that iteration returns v. Let $\sharp$*selects* denote the total number of *select* steps over the whole execution, i.e., the number of iterations of the main loop of the algorithm.

LEMMA 5.5. *The running time of the strongly polynomial algorithm is* $O(nm + \sharp selects \cdot \log n)$.

Proof. The total time for operations on the *d_heap* is $O(\sharp selects \cdot \log n)$, because there are $O(\sharp selects)$ operations on the *d_heap*. Next, consider operations on the *e_heap* and observe that we need not insert the vertex processed by a *select* step into the *e_heap* (if it belongs there) until the end of the *select* step. A saturating push over an edge (v, w) therefore causes one *increase key* operation, and if $e(w)$ becomes $\geq \Delta$, then w must be deleted from the *e_heap* and inserted into the *d_heap*. The number of *e_heap* delete operations at most equals the number of *d_heap* insert operations, which is $O(\sharp selects)$. The number of *increase key* operations is bounded by $\sharp$*selects* plus the number of saturating pushes, which is $O(nm)$. The total time needed for operations on the *e_heap* is therefore $O(nm + \sharp selects \cdot \log n)$.

Consider the remaining running time, excluding the time for heap operations. The initialization can be done in $O(m)$ time; in particular, note that random permutations can be computed in linear time (see, e.g., [S77]). Each iteration of the main loop runs in time $O(\log n)$, plus $O(\log n)$ times the number of *cut* operations executed, plus the time for executing saturating pushes over edges not in the dynamic trees data structure, plus the time for maintaining current edges. The total number of *cut* operations executed is $\leq \sharp selects$, because there is a distinct *link* operation corresponding to each *cut*, and the number of *link* operations is clearly $\leq \sharp selects$, since each iteration of the main loop executes at most one *link*. The overall time for maintaining the current edges is $O(nm)$, and the overall time for saturating pushes that are not associated with *cut* operations is $O(nm)$. Thus, the total running time is $O(nm + \sharp selects \cdot \log n)$. □

LEMMA 5.6. *$\sharp selects = O(nm)$, and the worst-case running time of the strongly polynomial algorithm is $O(nm \log n)$.*

Proof. It is easily seen that the total number of *cut* and *link* operations is $O(nm)$. It follows that the number of iterations of the main loop that execute either a *cut* or a *link* or a *relabel* or a saturating push is $O(nm + n^2) = O(nm)$.

To handle the remaining *select* steps, call each such step a *neat select* step and note that each neat *select* step moves flow from a nonroot in F to a root. Define Φ as the sum over all nonroot vertices v of $\phi(v)$, where ϕ is the function of the proof of Lemma 3.6, i.e.,

$$\phi(v) = \begin{cases} 0, & \text{if } e(v) = 0, \\ 1, & \text{if } 0 < e(v) \leq 2\Delta, \\ 2, & \text{if } e(v) > 2\Delta. \end{cases}$$

By the previous observation, each neat *select* step decreases Φ by at least 1. On the other hand, only saturating pushes and link operations can increase Φ, and each such operation increases Φ by at most 2. Since $\Phi = 0$ initially, the number of neat *select* steps is $O(nm)$, and the same bound applies to the total number of *select* steps.

The bound on the running time now follows from the previous lemma. □

The crucial part of the analysis is to show that for sufficiently dense graphs, randomly permuting the adjacency lists causes $\sharp selects$ to become significantly less than $\Theta(nm)$ with high probability. In the next section we analyze the efficiency of randomly permuting the adjacency lists, and based on this in §7 we give a simple analysis of $\sharp selects$.

6. PTR events. A *premature target relabeling* event (PTR event) is defined to be the relabeling of the head w of a current edge (v, w). In other words, a PTR event may be identified with a triple (v, w, k), where $0 \leq k \leq 2n - 2$ and $(v, w) \in E$ is the current edge of $v \in V$ when the vertex w, which currently has $d(w) = k$, gets relabeled. By definition, every *cut* executed by the *relabel* procedure corresponds to a PTR event; however, there may be other PTR events besides these, since the dynamic trees data structure may not contain all current edges. Denote by $\sharp$ptr the total number of PTR events.

The significance of PTR events is that a vertex changes its current edge if and only if either a saturating push occurs over the edge or a PTR event occurs on the edge. Since no forward huge edge is ever saturated (see §3), a vertex whose current edge is a forward huge edge changes its current edge exactly when a PTR event occurs on the edge.

LEMMA 6.1. *Over the whole execution, $\sharp huge\ sat\ pushes \leq m + \sharp$ptr.*

Proof. Between two consecutive saturating pushes over a reverse huge edge (w, v), there must be a push over the forward huge edge (v, w). When the push over (v, w) is executed, then $(v, w) \in F$. Consequently, between this step and the next saturating push over (w, v), a *cut* on (v, w) must be executed. Hence $\sharp huge\ sat\ pushes$ is at most m plus the number of cuts on forward huge edges, which is $\leq m + \sharp$ptr. □

It is easy to bound ♯ptr by $O(nm)$. However, for sufficiently dense graphs a much tighter bound can be obtained by making use of the fact that each vertex randomly and independently permutes its adjacency list at the start of the execution. Before delving into the analysis, we introduce some notation whose usefulness will become evident below.

For every finite set A, let $\mathrm{Perm}(A)$ be the set of all permutations of A, i.e., of all bijections from $\{1, \ldots, |A|\}$ to A. 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 $\mu(1), \ldots, \mu(|A|)$ and $\sigma(1), \ldots, \sigma(|B|)$. Given l permutations $\mu_1, \ldots, \mu_l$ of subsets of a finite set A, let $\Lambda(\mu_1, \ldots, \mu_l) = \max_{\sigma \in \mathrm{Perm}(A)} \sum_{i=1}^{l} \lambda(\mu_i, \sigma)$; note that this quantity does not depend on A. We call $\Lambda(\mu_1, \ldots, \mu_l)$ the *external coascent* of $\mu_1, \ldots, \mu_l$.

Example. This example is meant to familiarize the reader with the definitions of λ and Λ. For the duration of the example, we identify a permutation μ of a set B with the string $\mu(1) \ldots \mu(|B|)$ and use as our universe the set $A = \{a, b, c, d, e, f\}$ of six symbols.

Let $\mu = bead$ and $\sigma = fbadec$. Then $\lambda(\mu, \sigma) = 3$, since the sequence *bad* occurs in both μ and σ, whereas the only longer subsequence *bead* occurring in μ does not occur in σ.

Now take $\mu_1 = fade$, $\mu_2 = bead$ and $\mu_3 = dec$. Then $\Lambda(\mu_1, \mu_2, \mu_3) = 10$, since with $\sigma = fbadec$ we have $\sum_{i=1}^{3} \lambda(\mu_i, \sigma) = 10$, whereas it is not difficult to see that there is no permutation σ' of A with $\sum_{i=1}^{3} \lambda(\mu_i, \sigma') = 11$.

Identify V with the set $\{1, \ldots, n\}$ and let $\mu_v \in \mathrm{Perm}(\Gamma(v))$, for all $v \in V$. We shall say that the strongly polynomial algorithm is executed with the adjacency lists ordered according to $\mu_1, \ldots, \mu_n$ if the following holds after the initialization: For all $u \in V$ and all $v, w \in \Gamma(u)$, the edge (u, v) precedes (u, w) in u's adjacency list if and only if $\mu_u^{-1}(v) < \mu_u^{-1}(w)$. Furthermore, 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 all k with $0 \le k \le 2n-2$ and 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 now an execution of the algorithm with the adjacency lists ordered according to $\mu_1, \ldots, \mu_n$ that relabels according to $\sigma_0, \ldots, \sigma_{2n-2}$. Fix $v \in V$ and k with $0 \le k \le 2n-2$ and suppose that for some vertices $w_1, \ldots, w_l \in V$, the execution incurs PTR events $(v, w_1, k), \ldots, (v, w_l, k)$, in that order. Then, clearly $\mu_v^{-1}(w_1) < \cdots < \mu_v^{-1}(w_l)$ and $\sigma_k^{-1}(w_1) < \cdots < \sigma_k^{-1}(w_l)$, i.e., the sequences $\mu_v(1), \ldots, \mu_v(|\Gamma(v)|)$ and $\sigma_k(1), \ldots, \sigma_k(n)$ have a (not necessarily contiguous) common subsequence of length l, namely $w_1, \ldots, w_l$. It follows that for all $v \in V$ and all k with $0 \le k \le 2n-2$, the total number of PTR events of the form (v, w, k), where $w \in V$, is bounded by $\lambda(\mu_v, \sigma_k)$. Summing over all $v \in V$ for fixed k with $0 \le k \le 2n-2$, we see that the total number of PTR events of the form (v, w, k), where $v, w \in V$, is bounded by

$$\sum_{v \in V} \lambda(\mu_v, \sigma_k) \le \Lambda(\mu_1, \ldots, \mu_n).$$

A final summation over all values of k yields the next lemma.

LEMMA 6.2. *For all $v \in V$, let $\mu_v \in \mathrm{Perm}(\Gamma(v))$. If the strongly polynomial algorithm is executed with the adjacency lists ordered according to $\mu_1, \ldots, \mu_n$, then* ♯ptr $\le 2n \cdot \Lambda(\mu_1, \ldots, \mu_n)$.

LEMMA 6.3. *Let A be a finite set with $|A| = N$, let $A_1, \ldots, A_N$ be subsets of A and take $M = \sum_{i=1}^{n} |A_i|$. Suppose that μ_i is drawn randomly from the uniform distribution over* $\mathrm{Perm}(A_i)$, *for $i = 1, \ldots, N$, and that $\mu_1, \ldots, \mu_N$ are independent. Then for all $r \ge \sqrt{NM} + N \log N$, $\Lambda(\mu_1, \ldots, \mu_N) = O(r)$ with probability at least $1 - 2^{-r}$.*

Proof. Recall that $\Lambda(\mu_1, \ldots, \mu_N) = \max_{\sigma \in \mathrm{Perm}(A)} \psi(\sigma)$, where $\psi(\sigma) = \sum_{i=1}^{n} \lambda(\mu_i, \sigma)$. We will show the probability that $\psi(\sigma)$ is large to be very small for each fixed $\sigma \in \mathrm{Perm}(A)$. Multiplying that probability by the number of choices for σ, i.e., by $N!$, we obtain an upper bound on the probability that $\Lambda(\mu_1, \ldots, \mu_N)$ is large.

Hence, let $\sigma \in \mathrm{Perm}(A)$ be arbitrary but fixed. For $i = 1, \ldots, N$, let $\Lambda_i = \lambda(\mu_i, \sigma)$ and take $S = \psi(\sigma) = \sum_{i=1}^{n} \Lambda_i$, the quantity of interest. For $i = 1, \ldots, N$, denote $|A_i|$ by a_i. Assume $N \geq 2$.

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

$$\Pr(\Lambda_i \geq k) \leq \frac{\binom{a_i}{k}^2 (a_i - k)!}{a_i!} \leq \frac{a_i^k}{(k!)^2} \leq \left(\frac{e^2 a_i}{k^2}\right)^k.$$

It can be seen that Λ_i is unlikely to exceed $\sqrt{a_i}$ by very much. By applying the Cauchy–Schwarz inequality $|u \cdot v| \leq |u|\,|v|$ to the vectors $u = (1, \ldots, 1)$ and $v = (\sqrt{a_1}, \ldots, \sqrt{a_N})$, we obtain

$$\sum_{i=1}^{N} \sqrt{a_i} \leq \sqrt{N} \cdot \sqrt{\sum_{i=1}^{N} a_i} = \sqrt{NM}.$$

Hence $S = \sum_{i=1}^{N} \Lambda_i$ is unlikely to exceed $\sqrt{NM}$ by very much. We now establish a precise bound.

First observe that for arbitrary $x \in \mathbb{R}$,

$$\Pr(S \geq x) = e^{-x} e^{x} \Pr(e^S \geq e^x) \leq e^{-x} E(e^S).$$

Second, since $\mu_1, \ldots, \mu_N$ and hence also $e^{\Lambda_1}, \ldots, e^{\Lambda_N}$ are independent,

$$E(e^S) = E\left(e^{\sum_{i=1}^{N} \Lambda_i}\right) = E\left(\prod_{i=1}^{N} e^{\Lambda_i}\right) = \prod_{i=1}^{N} E(e^{\Lambda_i}).$$

We next bound the quantities $E(e^{\Lambda_i})$. Let $i \in \{1, \ldots, N\}$ and let $b_i \geq 0$ be an arbitrary integer. Then

$$E(e^{\Lambda_i}) = \sum_{k=0}^{\infty} e^k \Pr(\Lambda_i = k) \leq \sum_{k=0}^{b_i} e^k \Pr(\Lambda_i = k) + \sum_{k=b_i+1}^{\infty} e^k \Pr(\Lambda_i \geq k)$$

$$\leq e^{b_i} \sum_{k=0}^{b_i} \Pr(\Lambda_i = k) + \sum_{k=b_i+1}^{\infty} e^k \left(\frac{e^2 a_i}{k^2}\right)^k \leq e^{b_i} + \sum_{k=b_i+1}^{\infty} \left(\frac{e^3 a_i}{k^2}\right)^k.$$

Choose b_i to make $\frac{e^3 a_i}{k^2} \leq \frac{1}{2}$ for $k \geq b_i + 1$, i.e., take $b_i = \lfloor \sqrt{2e^3 a_i} \rfloor$. Then

$$E(e^{\Lambda_i}) \leq e^{b_i} + \sum_{k=b_i+1}^{\infty} 2^{-k} = e^{b_i} + 2^{-b_i} \leq 2e^{\sqrt{2e^3 a_i}}.$$

Putting together everything yields

$$\Pr(S \geq x) \leq e^{-x} E(e^S) = e^{-x} \prod_{i=1}^{N} E(e^{\Lambda_i}) \leq e^{-x} \prod_{i=1}^{N} (2e^{\sqrt{2e^3 a_i}})$$

$$= e^{-x} \cdot 2^N e^{\sqrt{2e^3} \sum_{i=1}^{N} \sqrt{a_i}} \leq 2^N e^{\sqrt{2e^3}\sqrt{NM} - x}.$$

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

$$\Pr(\Lambda(\mu_1, \ldots, \mu_N) \geq x) \leq N! \cdot 2^N e^{\sqrt{2e^3}\sqrt{NM} - x}$$

$$\leq e^{2N \log N + \sqrt{2e^3}\sqrt{NM} - x}.$$

Given any $r \geq \sqrt{NM} + N \log N$, now choose

$$x = 2N \log N + \sqrt{2e^3}\sqrt{NM} + r = O(r)$$

and observe that

$$\Pr(\Lambda(\mu_1, \ldots, \mu_N) \geq x) \leq 2^{-r}. \qquad \square$$

LEMMA 6.4. *For every* $\alpha \geq 1$, $\sharp\text{ptr} = O(\alpha \cdot (n^{3/2} m^{1/2} + n^2 \log n))$ *with probability at least* $1 - 2^{-\alpha(\sqrt{nm} + n \log n)}$.

Proof. Combine Lemmas 6.2 and 6.3 $\square$

7. A simple analysis of operations on nonhuge edges and of the overall running time. In order to complete the analysis of the running time, i.e., to bound $\sharp$*selects*, we have to show good bounds for the total number of operations on nonhuge edges. In this section we derive somewhat loose bounds by using a simple argument, and in the next section we give better bounds by using a more refined argument.

Let β be a real number with $2 \leq \beta = n^{O(1)}$ (we fix $\beta = 2 + \sqrt{m/n}$ to get the main result). Define the *status* of an undirected edge $\{v, w\}$ as follows:

$\{v, w\}$ is said to be *small* if $ucap(v, w) < \Delta/\beta$, *medium* if $\Delta/\beta \leq ucap(v, w) \leq 40n^2\Delta$, and (as defined in §3) *huge* if $ucap(v, w) > 40n^2\Delta$. Note that the status of an edge may change during the execution, but at most twice.

A push is called *small*, *medium*, or *huge*, respectively, if it is executed over a small edge, a medium edge, or a huge edge. We denote the number of medium saturating pushes by $\sharp$*med sat pushes*.

There is an obvious bound of $O(nm)$ of $\sharp$*med sat pushes*. However, for sufficiently dense graphs $\sharp$*med sat pushes* can be shown to be significantly less than proportional to nm. Intuitively, the reason for the relatively small number of saturating pushes over medium edges is that an edge is medium for only $O(\log n)$ phases, i.e., medium edges are "short-lived."

In order to bound the number of medium saturating pushes, we partition these into three classes, (a), (b), and (c). To define the partition, consider a medium saturating push over an edge (v, w). If either $d(v)$ or $d(w)$ has the same value at the time of the push as at a "phase boundary," i.e., at the beginning or at the end of a phase, then the push is a class (a) push. Otherwise, if the number of medium (undirected) edges incident with w at the time of the push is $\leq \deg(w)/\beta$, it is a class (b) push. If neither of these cases applies, the push is a class (c) push. Note that if a push of class (b) or (c) is executed over an edge (v, w) during a particular period of eligibility of (v, w), then that period of eligibility of (v, w) lies entirely within one phase.

LEMMA 7.1. *Over the whole execution, the number of class* (a) *and class* (b) *medium saturating pushes is* $O(nm/\beta + m \log n)$.

Proof. The number of class (a) pushes is $O(m \log n)$ because each edge is medium for only $O(\log n)$ phases, and between two successive saturating pushes over an edge both its tail vertex and its head vertex have to be relabeled.

The number of class (b) pushes is $O(nm/\beta)$. To see this, note that between two consecutive relabelings of a vertex w there are $\leq \deg(w)/\beta$ medium saturating pushes of class (b) into w. Summing over all $w \in V$ and all relabelings of w gives $O(\sum_{w \in V}(2n - 1) \deg(w)/\beta) = O(nm/\beta)$. □

The medium saturating pushes of class (c) are more difficult to handle, and we need a few more results before we can tally them.

Define the *throughput* of a saturating push over an edge (v, w) while $d(v) = k$ to be equal to $rescap(v, w)$ when $d(v)$ is set to k. In other words, the throughput is the total amount of flow sent over (v, w) during a maximal period of eligibility of (v, w). Note that the value of the saturating push may be less than its throughput.

LEMMA 7.2. *Over the whole execution, the number of nonsmall saturating pushes with throughput* $< \Delta/\beta$ *is* $\leq m + \sharp ptr$.

Proof. Consider a nonsmall saturating push over an edge (v, w) while $d(v) = k$. Since $ucap(v, w) \geq \Delta/\beta$, it follows that when $d(v)$ is set to k, then $rescap(w, v) > 0$; also, obviously, $rescap(v, w) > 0$. Assume that either (v, w) or (w, v) has been used as a current edge before this use of (v, w) as a current edge. It can then be seen that the previous use of (v, w) or (w, v) as a current edge was terminated by a PTR event. The lemma follows. □

The medium saturating pushes of class (c) that have throughput $< \Delta/\beta$ are easily taken care of by the previous lemma, so we now turn our attention to the remaining class (c) pushes. Recall that all the pushes contributing to the throughput of a class (c) push are executed in the same phase.

Consider a vertex w that is the head vertex of a medium saturating push of class (c). By the definition of class (c), a fraction of more than $1/\beta$ of the undirected edges incident with w are medium. The next lemma enables us to focus on such vertices and to give a sufficiently good bound on the number of class (c) saturating pushes with throughput $\geq \Delta/\beta$ into these vertices. The lemma is a generalization of Lemma 6 of [AO89], whose proof, with a minor modification, shows that the number of pushes with value $\geq \Delta$ in any phase is $O(n^2)$.

LEMMA 7.3. *For every subset* V' *of* V *and in every fixed* Δ*-phase, the number of pushes with value* $\geq \Delta$ *into the vertices of* V' (*i.e., pushes over edges of the form* (v, w), *where* $w \in V'$) *is* $O(|V'| \cdot n + D)$, *where* D *denotes the number of relabel operations executed in the phase.*

Since our main objective at this point is to provide intuition, we prove the lemma only for the special case when all the edge capacities are integers; the proof trivially extends to the case of rational edge capacities. A direct but less intuitive proof of the general case of the lemma is given in the next section (Lemma 8.3).

Suppose that all the edge capacities are integers, and also assume that the algorithm uses integer division, i.e., the occurrence of $\Delta/2$ in the procedure *select* is replaced by $\lfloor \Delta/2 \rfloor$. (Note that the test $\tilde{e}(v) < \Delta/2$ in the procedure *macropush* can be carried out without division.) Induction on the number of steps executed shows the preflow f, and hence also the flow excess e, to have integer values throughout the execution, and it can be seen that this modification does not affect our analysis of the algorithm. Furthermore, each unit of flow remains an indivisible entity throughout the execution. Call each unit of flow a *flow atom*. Related notions of flow atoms were previously used to analyze maximum-flow algorithms by, for example, Shiloach and Vishkin [SV82], Goldberg [G85], Cheriyan and Maheshwari [CM89], and Tunçel [T90].

Proof. (Lemma 7.3, integer edge capacities). The lemma is a straightforward consequence of a property of the so-called moving sequence of a flow atom. Consider a flow atom q and define $d(q)$ to be $d(v)$, where v is the vertex at which q is currently located. Focus on the movement of q during the phase under consideration and note that each change of $d(q)$ is caused by either a push operation or a relabel operation applied to the vertex at which q is currently located.

The *moving sequence* of q, $ms(q)$, is a string over the alphabet $\{\langle u, i\rangle : u \in V' \text{ and } 0 \leq i \leq 2n-1\} \cup \{\uparrow\}$ defined as follows: At the start of the phase, $ms(q)$ is empty. If q is pushed into a vertex $u \in V'$ (i.e., q is sent over an edge whose head vertex u is in V'), $\langle u, d(u)\rangle$ is appended to $ms(q)$, and if the vertex currently holding q undergoes a relabeling, then an $\uparrow$ is appended to $ms(q)$. Let $|ms(q)|_\uparrow$ denote the number of $\uparrow$ symbols in a moving sequence $ms(q)$, and let $|ms(q)|_\downarrow$ denote the number of remaining symbols in $ms(q)$, i.e., the number of symbols of the form $\langle u, i\rangle$. Finally let $n' = |V'|$.

The main property of moving sequences is

$$|ms(q)|_\downarrow \leq n' + |ms(q)|_\uparrow.$$

To show this relation, assume that $ms(q)$ contains at least one non-$\uparrow$ symbol, and let $\langle u, i\rangle$ be the first of these. Consider the prefix of $ms(q)$ up to and including the last symbol in $ms(q)$ with a first component of u and denote this prefix by $ms'(q)$. It is easy to show that $|ms'(q)|_\downarrow \leq 1 + |ms'(q)|_\uparrow$. To see this, note that each $\uparrow$ symbol corresponds to an increase of $d(q)$ by one, while each non-$\uparrow$ symbol corresponds to a decrease of $d(q)$ by one. Furthermore, by comparing the first non-$\uparrow$ symbol $\langle u, i\rangle$ in $ms'(q)$ with the symbol $\langle u, j\rangle$ at the end of $ms'(q)$ and noting that $d(u)$ is nondecreasing throughout the execution, we may conclude that $j \geq i$. Consequently, the number of non-$\uparrow$ symbols in $ms'(q)$ is at most one more than the number of $\uparrow$ symbols.

Now delete $ms'(q)$ from $ms(q)$. If the remaining string contains any non-$\uparrow$ symbols, let $\langle v, l\rangle$ be the first of these and consider the prefix $ms''(q)$ consisting of all symbols up to and including the last symbol with a first component of v. Using the same argument, it can be seen that $|ms''(q)|_\downarrow \leq 1 + |ms''(q)|_\uparrow$.

Repeating this argument a total of at most n' times shows that

$$|ms(q)|_\downarrow \leq n' + |ms(q)|_\uparrow.$$

Let Q denote the set of all flow atoms that are located at active vertices at the start of the phase (i.e., flow atoms located at s or t at the start of the phase are not in Q). Fact 3.4 clearly implies that $|Q| \leq 2n \cdot \Delta$. To count the number of pushes with value $\geq \Delta$ into the vertices of V', notice that each of these operations pushes $\geq \Delta$ flow atoms into some vertex $u \in V'$ and causes a symbol with a first component of u to be appended to the moving sequence of each of these flow atoms. Hence, the number of such pushes is

$$\begin{aligned}
&\leq (1/\Delta) \sum_{q \in Q} |ms(q)|_\downarrow \\
&\leq (1/\Delta) \sum_{q \in Q} (n' + |ms(q)|_\uparrow) \\
&\leq (1/\Delta)((2n \cdot \Delta \cdot n') + \sum_{q \in Q} |ms(q)|_\uparrow) \\
&\leq 2nn' + (1/\Delta)(3\Delta \cdot D) = 2nn' + 3D.
\end{aligned}$$

The last inequality follows because there are D *relabel* operations, and the total number of $\uparrow$ symbols appended by a single relabeling is at most 3Δ (Fact 3.4). □

We actually use a more technical but straightforward generalization of the previous lemma.

COROLLARY 7.4. *For every subset V' of V and in every fixed Δ-phase, the number of class* (c) *medium saturating pushes with throughput $\geq \Delta/\beta$ over edges of the form (v, w), where $w \in V'$, is $O(\beta(|V'| \cdot n + D))$, where D denotes the number of relabel operations executed in the phase.*

We need a technical definition. A Δ-phase is said to *hit* a vertex v if the number of medium (undirected) edges incident with v in the phase is $> \deg(v)/\beta$. Notice that each phase hits the head vertex of every class (c) medium saturating push executed in the phase. The following interpretation may be useful below: associate each undirected edge $\{v, w\}$ with the interval $[ucap(v, w)/(40n^2), \beta \cdot ucap(v, w)]$ on the x axis, and view "the phase" as a vertical line sweeping the x axis from ∞ to 0; the current value of Δ gives the current location of the sweep line, and an edge is medium exactly if its associated interval is currently intersected by the sweep line.

Fact 7.5. For all $v \in V$, the number of distinct phases that hit v is $O(\beta \log n)$. Therefore the sum over all phases of the number of vertices hit by the phase is $O(n\beta \log n)$.

Proof. Let the phases that hit v have parameters $\Delta_1, \Delta_2, \ldots$, and for $i = 1, 2, \ldots$, let $M_i(v)$ denote the set of medium (undirected) edges incident with v in the Δ_i-phase. The number of Δ_i- phases ($i \geq 2$) that hit v such that $M_1(v) \cap M_i(v) \neq \emptyset$ is $O(\log n)$, because in each such phase the parameter Δ_i is in the interval $[\Delta_1/(40n^2\beta), 40n^2\beta\Delta_1]$.

If v is hit by yet another phase (besides the $O(\log n)$ phases enumerated above) with parameter, say, $\Delta_j (j \geq 2)$, then $M_1(v) \cap M_j(v) = \emptyset$, and both $|M_1(v)|$ and $|M_j(v)|$ are $> \deg(v)/\beta$. Continuing the argument, it follows that v is hit by $O(\beta \log n)$ phases. □

LEMMA 7.6. *Over the whole execution, the number of class* (c) *medium saturating pushes is $O(n^2\beta^2 \log n + \sharp\text{ptr})$.*

Proof. By Lemma 7.2, the number of medium saturating pushes with throughput $< \Delta/\beta$ is $O(m + \sharp\text{ptr})$.

Consider a fixed Δ-phase and focus on the class (c) medium saturating pushes with throughput $\geq \Delta/\beta$ in that phase. Let V' denote the set of vertices hit by the phase, and let D denote the number of *relabel* operations in the phase. By applying Corollary 7.4, we can see that the number of class (c) medium saturating pushes with throughput $\geq \Delta/\beta$ is $O(\beta(|V'| \cdot n + D))$.

Finally, by summing over all phases and using Fact 7.5 and the fact that the total number of relabelings is $O(n^2)$, we find that the total number of class (c) medium saturating pushes with throughput $\geq \Delta/\beta$ is $O(n^2\beta^2 \log n)$. □

We are now ready to bound $\sharp$*selects* and thereby complete the simple analysis of the running time.

LEMMA 7.7. $\sharp selects = O(nm/\beta + n^2\beta^2 \log n + \sharp\text{ptr})$.

Proof. Every *cut* operation is associated with either a PTR event or a nonsmall saturating push, since no small edge is ever inserted into the dynamic trees data structure (Fact 5.2). Hence the number of *cut* operations is bounded by $\sharp$*med sat pushes* + $\sharp$*huge sat pushes* + $\sharp$ptr. The number of *link* operations clearly exceeds the number of *cut* operations by at most $n - 1$ because the forest F never contains more than $n - 1$ edges, and the number of *relabel* operations is $O(n^2)$. Therefore the number of *select* steps that execute either a *cut* or a *link* or a *relabel* is $O(n^2 + \sharp$*med sat pushes* + $\sharp$*huge sat pushes* + $\sharp$ptr).

The remaining *select* steps can be partitioned into two classes: those that execute one or more saturating pushes, and those that do not execute any saturating push.

The number of *select* steps that execute a saturating push and do not execute any *cut*, *link*, or *relabel* is $O(nm/\beta + \sharp$*med sat pushes* + $\sharp$*huge sat pushes*), because if a *select* step executes only small saturating pushes, then it executes at least $\beta/2$ saturating pushes, since

the flow excess of the processed vertex decreases from $\geq \Delta$ to $< \Delta/2$ and each small push has value $< \Delta/\beta$.

Now consider the *select* steps that do not execute any *cut*, *link*, *relabel*, or saturating push, and call each such step a *neat select* step. (The argument here is similar to that in the proof of Lemma 5.6.) The vertex processed by a neat *select* step is a nonroot in F throughout that step. Consider a fixed vertex v and focus on the part of the execution between two consecutive nonneat *select* steps that process v, or after the last such *select* step. Suppose that v is processed by one or more neat *select* steps in this part of the execution. When the first of these neat *select* steps is executed, then $\Delta \leq e(v) < 3\Delta$; hence it follows by Fact 5.4 that $e(v)$ decreases to zero after at most two neat *select* steps that process v, and over the whole execution, this gives a number of neat *select* steps that is at most twice the number of nonneat *select* steps. Before yet another neat *select* step that processes v, $e(v)$ has to increase from zero to $\geq \Delta$. Further, by Fact 5.3, any increase of $e(v)$ is caused by saturating pushes into v. Consider these saturating pushes and their associated edges (u, v), where $u \in V$. If each of these edges (u, v) has $ucap(u, v) < \Delta/\beta$ when v is processed by the neat *select* step, then clearly there are $\geq \beta$ saturating pushes into v, and over the whole execution this case gives $O(nm/\beta)$ neat *select* steps. Otherwise, either there is at least one nonsmall saturating push into v or the status of one of these edges changes (from small to medium or huge) between the earliest of these saturating pushes and the neat *select* step that processes v. Over the whole execution, the former case gives $O(\sharp med\ sat\ pushes + \sharp huge\ sat\ pushes)$ neat *select* steps, and the latter case gives $O(m)$ neat *select* steps, because the total number of status changes of edges is $\leq 2m$.

Therefore, the total number of neat *select* steps is at most twice the total number of nonneat *select* steps, plus $O(m + nm/\beta + \sharp med\ sat\ pushes + \sharp huge\ sat\ pushes)$. Hence

$$\begin{aligned}\sharp selects &= O(nm/\beta + n^2 + \sharp med\ sat\ pushes + \sharp huge\ sat\ pushes + \sharp \text{ptr}) \\ &= O(nm/\beta + n^2\beta^2 \log n + \sharp \text{ptr}).\end{aligned}$$

For the second equation, we use the bound on $\sharp med\ sat\ pushes$ given by Lemmas 7.1 and 7.6 and the bound on $\sharp huge\ sat\ pushes$ given by Lemma 6.1. □

LEMMA 7.8. *For every $\alpha \geq 1$, the strongly polynomial maximum-flow algorithm runs in time $O(\alpha \cdot (nm + n^{4/3}m^{2/3}(\log n)^{4/3} + n^2(\log n)^2))$ with probability at least $1 - 2^{-\alpha(\sqrt{nm} + n\log n)}$.*

Proof. By taking $\beta = 2 + (m/(n \log n))^{1/3}$ and using the previous lemma and the bound on $\sharp$ptr given in Lemma 6.4, we see that $\sharp selects = O(n^{4/3}m^{2/3}(\log n)^{1/3} + \alpha \cdot (n^{3/2}m^{1/2} + n^2 \log n))$ with probability at least $1 - 2^{-\alpha(\sqrt{nm} + n\log n)}$. The bound on the running time now follows from Lemma 5.5. □

For $m = \Omega(n(\log n)^4)$ the running time is $O(nm)$, with high probability.

8. A strengthened analysis of operations on medium edges and of the overall running time. An examination of the analysis in the previous section shows that the bottleneck in bounding $\sharp selects$ is the contribution due to $\sharp med\ sat\ pushes$, since our bound for $\sharp med\ sat\ pushes$ is $O(n^{4/3}m^{2/3}(\log n)^{1/3} + \sharp \text{ptr})$, whereas the bound for both $\sharp huge\ sat\ pushes$ and $\sharp$ptr is $O(n^{3/2}m^{1/2} + n^2 \log n)$. In this section, a tighter analysis of the contribution of $\sharp med\ sat\ pushes$ to $\sharp selects$ is developed.

Recall that the medium saturating pushes are partitioned into three classes, (a), (b), and (c), and that the number of pushes in these classes are $O(m \log n)$, $O(nm/\beta)$, and $O(n^2\beta^2 \log n + \sharp \text{ptr})$, respectively, for any β with $2 \leq \beta = n^{O(1)}$. The bottleneck term in the bound for $\sharp med\ sat\ pushes$ is due to the class (c) pushes.

In order to bound the contribution of the class (c) pushes to $\sharp selects$, we partition this class into two subclasses. A push over an edge (v, w) is called *terminal* if at the time of the push

$$|\{u : u \in V' \text{ and } d(u) = d(w)\}| < \beta,$$

where V' denotes the set of vertices hit by the current Δ-phase, i.e., if at the time of the push there are fewer than β "hit vertices" with labels equal to $d(w)$. Otherwise the push is *nonterminal*.

LEMMA 8.1. *Over the whole execution, the number of terminal pushes of class* (c) *is* $O(n^2\beta)$.

Proof. Fix $v \in V$ and $k \in \{1, \ldots, 2n-1\}$ and recall that if there are any class (c) pushes out of v at all while $d(v) = k$, then all pushes out of v while $d(v) = k$ occur in the same phase, i.e., the set V' does not change between the first and the last of these pushes. Since an edge out of v can become eligible only during a relabeling of v, while at the time of a terminal push out of v there are fewer than β eligible edges of the form (v, w), where $w \in V'$, it is now clear that the number of terminal class (c) pushes out of v while $d(v) = k$ is bounded by β. Summing over all $v \in V$ and all $k \in \{1, \ldots, 2n-1\}$ yields a total of $O(n^2\beta)$ terminal class (c) pushes. □

The next lemma is the key instrument for bounding the contribution of the nonterminal class (c) pushes to $\sharp$*selects*.

LEMMA 8.2. *The sum over all Δ-phases of the ratio of the total value of all nonterminal pushes in the Δ-phase to $\Delta/2$ is $O(n^2 \log n)$.*

To see the relevance of this lemma, notice that it directly gives an $O(n^2 \log n)$ bound on the number of nonterminal pushes with value $\geq \Delta/2$. To prove Lemma 8.2, we need a stronger version of Lemma 7.3. The following lemma may be used to bound the total value of nonterminal pushes in any Δ-phase by taking V' (in the lemma) to be the set of vertices hit by the phase and taking γ (in the lemma) to be equal to β.

LEMMA 8.3. *Let γ be a number ≥ 1. For every subset V' of V and in every fixed Δ-phase, the total value of pushes into vertices w such that when the push is executed*

$$|\{u : u \in V' \text{ and } d(u) = d(w)\}| \geq \gamma$$

is $O((|V'| \cdot n/\gamma + D)\Delta)$, where D denotes the number of relabel operations executed in the phase.

Proof. Let $h = |V'|$ and $V' = \{v_1, \ldots, v_h\}$ and for all $v \in V$, define the *fooling height* of v as

$$d'(v) = \max_{i_1 \geq d(v_1), \ldots, i_h \geq d(v_h)} |\{k \in \mathbb{N}_0 : 0 \leq k < d(v) \text{ and } |\{j : i_j = k\}| \geq \gamma\}|.$$

Intuitively, $d'(v)$ counts the maximum number of "dense virtual distance levels" between v and t, where a vertex $v_j \in V'$ 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 in V'.

d' has the following properties:

(1) $\forall v \in V : 0 \leq d'(v) \leq h/\gamma$;

(2) $\forall v, w \in V : d(v) > d(w) \Longrightarrow d'(v) \geq d'(w)$;

(3) $\forall v, w \in V : (d(v) > d(w) \text{ and } |\{u \in V' : d(u) = d(w)\}| \geq \gamma) \Longrightarrow d'(v) > d'(w)$;

(4) a relabeling of a vertex $v \in V$ increases $d'(v)$ by at most 1 and does not increase $d'(w)$ for any $w \in V \backslash \{v\}$.

Define the potential function

$$\Phi = \sum_{v \in V - \{s,t\}} e(v) \cdot d'(v).$$

At the start of the Δ-phase, $\Phi \leq 2n\Delta \cdot h/\gamma$ (by property (1) and Fact 3.4), and $\Phi \geq 0$ always. Φ does not increase due to push operations (by property (2)), and a relabeling increases Φ by at most 3Δ (by property (4) and Fact 3.4). It follows that the total increase in Φ during the Δ-phase is at most $3\Delta \cdot D$. Consequently, the total decrease in Φ during the Δ-phase is at most $2n\Delta \cdot h/\gamma + 3\Delta \cdot D$. Finally note that each push satisfying the condition of the lemma and of value c causes Φ to decrease by at least c (by property (3)). □

Proof (Lemma 8.2). Consider a fixed Δ-phase. Let V' denote the set of vertices hit by the phase, let D denote the number of relabelings executed in the phase, and take $\gamma = \beta$. Now apply Lemma 8.3, noting that every nonterminal push satisfies the condition of the lemma. Hence, the ratio of the total value of all nonterminal pushes in the phase to $\Delta/2$ is $O(|V'| \cdot n/\beta + D)$.

The lemma now follows by summing over all phases, using Fact 7.5 to bound the sum over all phases of the number of vertices hit by the phase and noting that the total number of *relabel* operations is $O(n^2)$. □

The proof of the next lemma follows the same outline as that of Lemma 7.7.

LEMMA 8.4. $\sharp selects = O(nm/\beta + n^2\beta + n^2 \log n + \sharp \text{ptr})$.

Proof. First we give an improved bound on the number of *select* steps that execute either a *cut* or a *link* or a *relabel*, by giving an improved bound on the number of *cut* operations. Notice that the throughput of any saturating push associated with a *cut* operation is $\geq \Delta/2$, because whenever an edge (v, w) is inserted into F (the procedure *macropush*), then $rescap(v, w) \geq \Delta/2$ (by Fact 5.2). We claim that the number of medium saturating pushes with throughput $\geq \Delta/2$ is $O(nm/\beta + n^2\beta + n^2 \log n)$. To see this, focus on the nonterminal class (c) pushes with throughput $\geq \Delta/2$, since the total number of class (a), class (b), and terminal class (c) pushes is $O(nm/\beta + m \log n + n^2\beta)$, by Lemmas 7.1 and 8.1. By applying Lemma 8.2 it can be seen that the number of nonterminal class (c) pushes with throughput $\geq \Delta/2$ is $O(n^2 \log n)$. The claim follows. Consequently, the number of *cut* operations is $O(nm/\beta + n^2\beta + n^2 \log n + \sharp \textit{huge sat pushes} + \sharp \text{ptr}) = O(nm/\beta + n^2\beta + n^2 \log n + \sharp \text{ptr})$, and the number of *select* steps that execute either a *cut* or a *link* or a *relabel* is also $O(nm/\beta + n^2\beta + n^2 \log n + \sharp \text{ptr})$.

The remaining *select* steps are partitioned into two classes: those that execute one or more saturating pushes, and those that do not execute any saturating push. (The argument here is a refined version of that used in the proof of Lemma 7.7.)

The number of *select* steps that execute a saturating push and do not execute any *cut*, *link*, or *relabel* is $O(nm/\beta + n^2\beta + n^2 \log n + \sharp \textit{huge sat pushes})$. To see this, first focus on such a *select* step that executes a nonterminal class (c) push. Notice that all of the saturating pushes executed by the *select* step are nonterminal and that the total value of these pushes is $\geq \Delta/2$. Now, by applying Lemma 8.2, we see that over the whole execution, this case gives $O(n^2 \log n)$ *select* steps. Over the whole execution, the number of *select* steps that execute either a class (a), a class (b), or a terminal class (c) push is $O(nm/\beta + m \log n + n^2\beta)$, and the number of *select* steps under consideration that execute only small saturating pushes is $O(nm/\beta)$, because any such step executes at least $\beta/2$ saturating pushes, since the flow excess of the processed vertex decreases from $\geq \Delta$ to $< \Delta/2$ and since each small push has value $< \Delta/\beta$.

Now consider the *select* steps that do not execute any *cut*, *link*, *relabel*, or saturating push, and call each such step a *neat select* step. As noted before, the vertex v processed by a neat *select* step is a nonroot in F throughout that step. Consider a fixed vertex v and focus on the part of the execution between two consecutive nonneat *select* steps that process v, or after the last such *select* step. Suppose that v is processed by one or more neat *select* steps in this part of the execution. When the first of these neat *select* steps is executed, then $\Delta \leq e(v) < 3\Delta$. Hence, it follows by Fact 5.4 that $e(v)$ decreases to zero after at most two neat *select* steps that process v, and over the whole execution, this gives a number of neat *select* steps that is at most

twice the number of nonneat *select* steps. Before yet another neat *select* step that processes v, $e(v)$ has to increase from zero to $\geq \Delta$. Furthermore, by Fact 5.3, any increase of $e(v)$ must be caused by saturating pushes into v. Consider these saturating pushes and their associated edges (u, v), where $u \in V$. If each of these edges (u, v) has $ucap(u, v) < \Delta/\beta$ when v is processed by the neat *select* step, then clearly there are $\geq \beta$ saturating pushes into v, and over the whole execution this case gives $O(nm/\beta)$ neat *select* steps. Otherwise, either the status of one of these edges changes (from small to medium or huge) between the earliest of these saturating pushes and the neat *select* step that processes v, or there is at least one nonsmall saturating push into v. Over the whole execution, the former case gives $O(m)$ neat *select* steps, because the total number of status changes of edges is $\leq 2m$. If there is an increase of $e(v)$ from zero to $\geq \Delta$ due to a nonsmall saturating push and zero or more small saturating pushes, followed by a neat *select* step that processes v, then we have two mutually exclusive cases. Either the increase of $e(v)$ is due to one or more nonterminal class (c) pushes whose values sum to $\geq \Delta/2$ together with zero or more small saturating pushes, or the increase of $e(v)$ is due to one or more of the following together with zero or more small saturating pushes: one or more nonterminal class (c) pushes whose values sum to $< \Delta/2$ together with $\geq \beta/2$ small saturating pushes, a class (a) push, a class (b) push, a terminal class (c) push, or a huge saturating push. Over the whole execution, the first case gives $O(n^2 \log n)$ neat *select* steps (by Lemma 8.2), and the second case gives $O(nm/\beta + m \log n + n^2\beta + \sharp$*huge sat pushes*$)$ neat *select* steps. Therefore the total number of neat *select* steps is at most twice the total number of nonneat *select* steps plus $O(nm/\beta + n^2\beta + n^2 \log n + \sharp$*huge sat pushes*$)$.

Finally, the number of *select* steps, $\sharp$*selects*, is $O(nm/\beta + n^2\beta + n^2 \log n + \sharp\text{ptr})$, using the bound on $\sharp$*huge sat pushes* given by Lemma 6.1. □

THEOREM 8.5. *For every* $\alpha \geq 1$, *a maximum flow can be computed in time* $O(\alpha \cdot (nm + n^2(\log n)^2))$ *with probability at least* $1 - 2^{-\alpha(\sqrt{nm}+n\log n)}$.

Proof. By taking $\beta = 2 + \sqrt{m/n}$ and using the previous lemma and the bound on $\sharp$ptr given in Lemma 6.4, we see that $\sharp$*selects* $= O(\alpha \cdot (n^{3/2}m^{1/2} + n^2 \log n))$ with probability at least $1 - 2^{-\alpha(\sqrt{nm}+n\log n)}$. The bound on the running time follows from Lemma 5.5. □

Notice that for $m = \Omega(n(\log n)^2)$ the running time is $O(nm)$, with high probability.

Remark. If, as in [CH89], a new random permutation μ_v of $\Gamma(v)$ is computed at each relabeling of v, then the failure probability of Theorem 8.5 can be reduced even further to $2^{-\alpha(n^{3/2}m^{1/2}+n^2\log n)}$.

9. Conclusion. We have shown that by using randomization on nonsparse graphs (i.e., $m = \Omega(n(\log n)^2)$) we can compute a maximum flow in $O(nm)$ time with high probability. The crucial new idea in our analysis is the notion of PTR events.

The preliminary version of this work [CH89] stimulated new research that has resulted in further advances on computing maximum flows. The original analysis is slightly loose, giving an $O(nm + n^{3/2}m^{1/2}(\log n)^{3/2} + n^2(\log n)^2)$ running time and an $O(n^{3/2}m^{1/2}(\log n)^{1/2} + n^2 \log n)$ bound on $\sharp$*selects*, both with high probability. Tarjan [T89] improved the analysis to obtain a running time of $O(nm + n^2(\log n)^2)$, with high probability. The improved analysis presented in §8, which differs from the analysis of [T89], was discovered subsequently and reported briefly in [CHM90].

Alon [A90] presented a simple way of derandomizing the strongly polynomial algorithm without affecting the running time for sufficiently dense graphs (i.e., $m = \Omega(n^{5/3} \log n)$): at initialization, the adjacency lists of the network vertices are permuted according to "pseudo-random permutations" (i.e., a set $\{\xi_1, \ldots, \xi_n\}$ of permutations of V with $\Lambda(\xi, \ldots, \xi_n) \ll n^2$) that are generated deterministically in $O(n^2)$ time. The running time of the resulting deterministic algorithm is $O(nm + n^{8/3} \log n)$. Recently, faster deterministic algorithms were described in [KRT93] and [PW93].

Another paper based on the present one is [CHM90], which shows that a maximum flow can be computed in $O(n^3/\log n)$ time on a uniform-cost RAM, and that the number of operations executed on flow variables can be improved from $O(nm)$ for the strongly polynomial algorithm here to $O(n^{8/3}(\log n)^{4/3})$.

One avenue for further research suggested by this work is to investigate whether randomization is useful for solving related problems on networks such as the minimum-cost flow problem and the maximum-weight matching problem.

REFERENCES

[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.

[A90] 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 *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, 1989, pp. 118–123.

[CHM90] J. Cheriyan, T. Hagerup, and K. Mehlhorn, *Can a maximum flow be computed in o(nm) time?*, in *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, 1990, Lecture Notes in Comput. Sci., 443 Springer-Verlag, New York, pp. 235–248.

[CM89] J. Cheriyan and S. N. Maheshwari, *Analysis of preflow push algorithms for maximum network flow*, SIAM J. Comput., 18 (1989), pp. 1057–1086.

[D70] E. A. Dinic, *Algorithm for solution of a problem of maximum flow in a network with power estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277–1280.

[EK72] J. Edmonds and R. M. Karp, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach., 19 (1972), pp. 248–264.

[FF57] L. R. Ford and D. R. Fulkerson, *A simple algorithm for finding maximal network flows and an application to the Hitchcock problem*, Canad. J. Math., 9 (1957), pp. 210–218.

[FF62] ———, *Flows in Networks*, Princeton Univ. Press, Princeton, NJ, 1962.

[FT87] M. L. Fredman and R. E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. Assoc. Comput. Mach., 34 (1987), pp. 596–615.

[F86] S. Fujishige, *A capacity-rounding algorithm for the minimum-cost circulation problem: A dual framework of the Tardos algorithm*, Math. Programming, 35 (1986), pp. 298–308.

[Ga85] H. N. Gabow, *Scaling algorithms for network problems*, J. Comput. Systems Sci., 31 (1985), pp. 148–168.

[GaT88] Z. Galil and E. Tardos, *An $O(n^2(m + n\log n)\log n)$ min-cost flow algorithm*, J. Assoc. Comput. Mach., 35 (1988), pp. 374–386.

[G85] A. V. Goldberg, *A new max-flow algorithm*, Tech. Rep. MIT/LCS/TM-291, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1985.

[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] ———, *Finding minimum-cost circulations by canceling negative cycles*, J. Assoc. Comput. Mach., 36 (1989), pp. 873–886.

[GT90] ———, *Finding minimum-cost circulations by successive approximation*, Math. Oper. Res., 15 (1990), pp. 430–466.

[GTT90] A. V. Goldberg, E. Tardos, and R. E. Tarjan, *Network flow algorithms*, in *Paths, Flows, and VLSI-Layout*, Algorithms and Combinatorics 9, B. Korte, L. Lovász, H. J. Prömel and A. Schrijver, eds., Springer-Verlag, Berlin, 1990, pp. 101–164.

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

[K74] A. V. Karzanov, *Determining the maximal flow in a network by the method of preflows*, Soviet Math. Dokl., 15 (1974), pp. 434–437.

[KRT93] V. King, S. Rao, and R. Tarjan, *A faster deterministic maximum flow algorithm*, preprint, January 1993; preliminary version in *Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms*, (1992), pp. 157–164; J. Algorithms, 17 (1994), pp. 447–474.

[O88] J. B. ORLIN, *A faster strongly polynomial minimum cost flow algorithm*, in Proceedings of the 20th ACM Symposium on Theory of Computing, 1988, pp. 377–387. Also in Sloan Working Paper No. 3060-89-MS, Massachusetts Institute of Technology, Cambridge, MA, 1989; Oper. Res., 41 (1993), pp. 338–350.

[PW93] S. PHILLIPS AND J. WESTBROOK, *Online load balancing and network flow*, in Proceedings of the 25th ACM Symposium on Theory of Computing, 1993, pp. 402–411.

[S77] R. SEDGEWICK, *Permutation generation methods*, Comput. Surv., 9 (1977), pp. 137–164.

[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.

[SV82] Y. SHILOACH AND U. VISHKIN, *An $O(n^2 \log n)$ Parallel MAX-FLOW Algorithm*, J. Algorithms, 3 (1982), pp. 128–146.

[T85] E. TARDOS, *A strongly polynomial minimum cost circulation algorithm*, Combinatorica, 5 (1985), pp. 247–255.

[T83] R. E. TARJAN, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[T89] ———, Personal communication, September 1989.

[T90] L. TUNÇEL, *On the complexity of preflow-push algorithms for maximum-flow problems*, Tech. Rep. 901, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY, April 1990; Algorithmica, 11 (1994), pp. 353–359.