Can a Maximium Flow be Computed in $o(nm)$ Time?

Joseph Cheriyan
Torben Hagerup
Kurt Mehlhorn

A 90/07

May 1990

FB 14, Informatik
Universität des Saarlandes
D–6600 Saarbrücken
West Germany

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 $O(n^{3/2}m^{1/2}(\log n)^{3/2} + n^2(\log n)^2)$ flow operations with high probability.

Specializing to the case in which all capacities are integers bounded by U, we show that a maximum flow can be computed using $O(n^{3/2}m^{1/2} + n^2(\log U)^{1/2})$ flow operations. Finally, we argue that several of our results yield optimal parallel algorithms.

This research was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM).

Can a Maximum Flow be Computed in $o(nm)$ Time?

JOSEPH CHERIYAN, TORBEN HAGERUP, AND KURT MEHLHORN

Fachbereich Informatik, Universität des Saarlandes
D–6600 Saarbrücken, West Germany

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 $O(n^{3/2}m^{1/2}(\log n)^{3/2} + n^2(\log n)^2)$ flow operations with high probability.

Specializing to the case in which all capacities are integers bounded by U, we show that a maximum flow can be computed using $O(n^{3/2}m^{1/2} + n^2(\log U)^{1/2})$ flow operations. Finally, we argue that several of our results yield optimal parallel algorithms.

1. Introduction

The fastest deterministic and randomized algorithms for computing a maximum flow in a network with n vertices and m edges have running times of $O(\min\{nm \log n, nm + n^{8/3} \log n\})$ (Alon's [Al89] derandomization of [CH89]) and $O(\min\{nm \log n, nm + n^2(\log n)^2\})$ (Tarjan's [Ta89] improved analysis of [CH89]), respectively. 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 bound U on the maximum capacity 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 answers the question posed in the title in the affirmative.

Our algorithm is based on earlier work in [CH89], [GT88], and [AO87], all of which in turn uses the generic maximum-flow algorithm of Goldberg and Tarjan [GT88], which works by manipulating a so-called *preflow* 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 on 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 $O(n^{8/3}(\log n)^{4/3})$. All previous algorithms execute $\Omega(nm)$ flow operations. Using randomization, we can do even better: A maximum flow can be computed using $O(n^{3/2}m^{1/2}(\log n)^{3/2} + n^2(\log n)^2)$ flow operations with

---

This research was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM).

high probability. In fact, our deterministic algorithm is obtained from the randomized algorithm by applying a derandomization technique due to Alon [Al89].

The bottleneck in our algorithms turns out to be a simple combinatorial problem on (unweighted) graphs, 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 algorithm. The idea behind our improvement of this bound for dense networks, by a factor of $\Theta(\log n)$, is to represent the residual graph by its adjacency matrix and to partition the matrix into $1 \times \lfloor \log n \rfloor$ submatrices. This enables us to process a submatrix in constant time by table look-up while searching for a current edge. This method depends critically on the use of the (standard) uniform cost measure for defining running time.

For networks with integer capacities, we give an incremental algorithm based on the excess scaling algorithm of Ahuja and Orlin [AO87]. The algorithm is simple; however, its analysis hinges on a nontrivial potential function. We show that the number of flow operations is $O(n^{3/2}m^{1/2} + n^2 \log U)$. Using the wave scaling technique of [AOT89], the number of flow operations can be reduced to $O(n^{3/2}m^{1/2} + n^3(\log U)^{1/2})$ We mention that our use of "visible excesses" in some ways is similar to the use of "available excesses" in [AOT89].

The generic incremental algorithm is introduced in Section 3, and the algorithm for integer networks is described in Section 4. Section 5 discusses the current edge problem. The strongly polynomial algorithm is presented in Section 6 and analysed in Sections 7–9.

2. Definitions and notation

For any set V and any $e = (v, w) \in V \times V$, 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. For any $E \subseteq V \times V$, denote by $\overline{E}$ the closure of E under reversal, i.e., $\overline{E} = E \cup \{rev(e) : e \in E\}$. Further, for any function $\phi : E \to \mathbb{R}$, let $\overline{\phi} : \overline{E} \to \mathbb{R}$ be given by $\overline{\phi}(e) = \phi(e)$ for $e \in E$, $\overline{\phi}(e) = 0$ for $e \in \overline{E} \backslash E$. A *network* is a tuple $G = (V, E, cap, s, t)$, where (V, E, cap) is an edge-weighted directed graph, $cap : E \to \mathbb{R}_+ \cup \{0\}$, and s and t are distinct vertices in V. A *preflow* in G is a function $f : \overline{E} \to \mathbb{R}$ with the following properties:

(1) $f(rev(e)) = -f(e)$, for all $e \in \overline{E}$ (antisymmetry constraint);
(2) $f(e) \leq \overline{cap}(e)$, for all $e \in \overline{E}$ (capacity constraint);
(3) $\sum_{e \in \overline{E},\ head(e) = v} f(e) \geq 0$, for all $v \in V \backslash \{s\}$ (nonnegativity constraint).

A preflow f in G is a *flow* if $\sum_{e \in \overline{E},\ head(e) = v} f(e) = 0$ for all $v \in V \backslash \{s, t\}$ (flow conservation constraint). The *value* of f is $\sum_{e \in \overline{E},\ head(e) = t} f(e)$, and a *maximum flow* in G is a flow in G of maximum value. An edge $e \in \overline{E}$ is *residual* (with respect to f) if $f(e) < \overline{cap}(e)$. A *push* on e of value $c \in \mathbb{R}$ is an increase in $f(e)$ by c. The push is *saturating* iff $f(e) = \overline{cap}(e)$ afterwards. 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 each edge $(v, w) \in \overline{E}$ that is residual with respect to f.

We use what we consider to be the traditional model for the study of problems on networks. Capacities and flow values are represented by *real numbers*, on which the only allowed arithmetical operations are addition and subtraction, and all other quantities are represented by *integers*, on which we allow addition, subtraction, multiplication and integer division. In addition, we assume for both data types standard operations for comparisons, data movement, the constant 1, etc. For n-vertex input networks, we allow integers of absolute value $n^{O(1)}$, and we charge constant time for each basic operation on real numbers or integers (uniform cost measure). In keeping with common usage, we employ the term "flow operation" to mean any operation on real numbers.

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 symmetric network $G = (V, E, cap, s, t)$. Let $n = |V|$ and $m = |E|$. In order to avoid trivialities, we assume $m \geq n \geq 3$. Let $V^+ = V \setminus \{s, t\}$. The main variables used by the incremental generic algorithm are

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

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

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

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

$$e(v) = \sum_{e \in \overline{E^*},\ head(e) = v} f(e) \qquad e^*(v) = \max\{e(v) - \sum_{e \in E \setminus E^*,\ tail(e) = v} cap(e), 0\}.$$

Although the functions e and e^* 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 by definition is antisymmetric, low-level flow manipulation is carried out by the procedure

```
procedure set flow(e: edge; c: real);
  f(e) := c; f(rev(e)) := - c;
```

with the special case

```
procedure saturate(e: edge);
  set flow(e, cap(e));
```

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

```
procedure push(e: edge; c: real);
```

Precondition: $e = (v, w) \in \overline{E^*}$, $v \in V^+$, e is eligible, and $0 < c \leq \min\{e^*(v), rescap(e)\}$.

```
  set flow(e, f(e) + c);
procedure relabel(v: vertex);
```

Precondition: $v \in V^+$, $e^*(v) > 0$, and no edge in $\overline{E^*}$ with tail v is eligible.

```
  d(v) := d(v) + 1;
procedure add edge(e: edge);
```

Precondition: $e \in E \setminus E^*$.

```
  E* := E* ∪ {e};
  if d(tail(e)) > d(head(e)) then saturate(e);
procedure generic initialize;
  for all e ∈ E do set flow(e,0); (* zero flow is default for new edges *)
  for all v ∈ V\{s} do d(v) := 0; d(s) := n;
  E* := ∅;
  for all e ∈ E with s ∈ {tail(e), head(e)} do add edge(e);
```

Incremental generic algorithm:

```
  generic initialize;
  while max{e(v) : v ∈ V+} > 0
  do execute some push, relabel or add edge operation
      whose precondition is satisfied. (* there always is one *)
```

An execution of *push*(e, c) and of *relabel*(v) is called a *push* on e and a *relabeling* of v, respectively. We next show the partial correctness of the algorithm and give a few additional properties. In stating invariants for the algorithm, we consider *set flow* 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.

Fact 1: For $v \in V$, let $h(v) = \sum_{e \in E \setminus E^*, tail(e)=v} cap(e)$. For all $v \in V$, if $e(v) \geq h(v)$ at some time during the execution, then $e(v) \geq h(v)$ forever after. In particular, for all $v \in V \setminus \{s\}$, $e(v) < h(v) \Rightarrow d(v) = 0$.

Lemma 3.1: At all times during an execution of the incremental generic algorithm,
(1) f is a preflow;
(2) d is a valid labeling.

Proof: (1) and (2) hold initially, and they are not invalidated by calls of *push* or *relabel* (cf. Lemma 3.1 of [GT88]). Furthermore calls of *add edge* are easily seen to preserve (2). The only remaining issue is that a saturating push over an edge (v, w) performed during a call of *add edge* might invalidate the nonnegativity constraint $e(v) \geq 0$. However, when the push takes place, $d(v) > d(w) \geq 0$, and it follows from Fact 1 that $e(v) \geq 0$ after the push. ∎

Lemma 3.2: Suppose that the algorithm terminates. Then, at termination, the extension $\tilde{f} : E \to \mathbb{R}$ of f with $\tilde{f}(e) = 0$ for all $e \in E \setminus E^*$ is a maximum flow in G.

Proof: At termination, f is a flow in G^*, so $\tilde{f}$ is a flow in G. If $\tilde{f}$ is not a maximum flow in G, it follows from Theorem 3.2 of [GT88] that there is a simple path p in G from s to t all of whose edges are residual with respect to $\tilde{f}$. Let v be the last vertex on p reachable from s over a path of edges on p belonging to E^*. If $v \neq t$, the edge on p with tail v belongs to $E \setminus E^*$, and Fact 1 implies that $d(v) = 0$, which is true for $v = t$ as well. But since $d(s) = n$ and v is reachable from s over a path of length $\leq n - 1$ all of whose edges are residual with respect to f, this contradicts the validity of d. ∎

Fact 2: An ineligible edge $(v, w) \in E$ can become eligible only during a relabeling of v.

Lemma 3.3: For all $v \in V$ and at all times during the execution, $d(v) \leq 2n - 1$. In particular, the total number of relabelings executed by the algorithm is $< 2n^2$.

Proof: See Lemmas 3.7 and 3.8 of [GT88]. ∎

4. The incremental excess scaling algorithm

Because it illustrates our main ideas in a very simple setting, we describe in this section an incremental excess scaling algorithm for the case where all edge capacities are integers bounded by U. The algorithm is an adaptation of the excess scaling algorithm of Ahuja and Orlin [AO87] to the incremental paradigm.

For each $e \in E^*$, define the *undirected capacity* of e as $ucap(e) = cap(e) + cap(rev(e))$. The execution is divided into *phases* parameterized by the value of a *scaling parameter* Δ. The algorithm repeatedly chooses a vertex $v \in V^+$ with $e^*(v) \geq \Delta$ 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 $e^*(v) \geq \Delta$, the current phase ends, Δ is replaced by $\Delta/2$, all edges $e \in E \setminus E^*$ with $ucap(e) \geq \Delta/\beta$ are added to E^*, and the next phase begins. Here $\beta \geq 1$ is an integer to be chosen later.

We assume E^* to be represented by a set of adjacency lists. For all $v \in V$, the first eligible edge in the adjacency list of v (if any) is called the *current edge* of v. The complete program follows.

function *ce*(*v*: *vertex*): *edge*;
Return the current edge of v, or *nil* if v has no current edge;

Incremental scaling algorithm:

```
generic initialize;
Δ := 2^⌊log_2 U⌋;
while Δ ≥ 1
do begin
     for all e ∈ E\E* with ucap(e) ≥ Δ/β do add edge(e);
     while max{e*(v) : v ∈ V+} ≥ Δ
     do begin
          Among the vertices v ∈ V+ with e*(v) ≥ Δ, choose v as one with minimal d(v);
          if cc(v) = nil
          then relabel(v)
          else push(cc(v), min{Δ, rescap(cc(e))});
        end;
     Δ := Δ/2;
   end.
```

The algorithm is easily seen to be an instance of the incremental generic algorithm, and hence to be partially correct. We now analyze its running time. Denote by $\sharp pushes$ the total number of pushes executed by the algorithm, and by T_{cc} the total time spent in the routine *cc*. T_{cc} will be analyzed in the following section.

Lemma 4.1: The algorithm uses $O(q)$ flow operations and $T_{cc} + O(q)$ time, where $q = \sharp pushes + n \log U + n^2 + m \log(n\beta)$.

Proof: Ahuja and Orlin [AO87] have described a simple implementation that allows the total time spent in testing the condition of the inner loop of the program and in choosing v to be bounded by $O(\sharp pushes + n \log U)$. Each relabeling takes constant time, and the total number of relabelings is $O(n^2)$ by Lemma 3.3. A single update of E^* following a decrease of Δ can be time-consuming. However, if the edges are initially sorted by their undirected capacities, the necessary time is $O(1)$ per edge added to E^*, for a total time of $O(m)$. Finally, there are $O(\log U)$ phases, each contributes $O(1)$ time that has not yet been accounted for, and $O(m \log(n\beta))$ time suffices for the initialization ($O(m \log n)$ time for sorting the edges, and $O(m \log \beta)$ time for multiplying their capacities by β). ∎

For $v \in V$ and $i = 1, 2, \ldots$, denote by $\deg_i(v)$ the number of edges with tail 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 $m_i = \sum_{v \in V} \deg_i(v)$ and denote by $\sharp relabels_i$ the number of relabelings carried out in phase i. The following observations are immediate:

Fact 3: Consider a push on an edge $e = (v, w)$ carried out during a phase (i.e., not in a call of *add edge*). At the time of the push, e is a current edge, the value of the push is $< 2\Delta$, and if $w \in V^+$, then $e^*(w) < \Delta$ immediately before the push.

Fact 4: For all $v \in V^+$ and for $i = 1, 2, \ldots$, $e^*(v) < 3\Delta + 2 \deg_i(v)\Delta/\beta$ throughout phase i.

Using arguments similar to those of [GT88] and [AO87], it is easy to bound the number of saturating pushes by $O(nm)$ and the number of nonsaturating pushes by $O(nm/\beta + n^2 \log U)$. In order to obtain a tighter bound on the number of saturating pushes, we define a push on an edge (u, v) to be *terminal* if $|\{w \in V : d(w) = d(v)\}| < \beta$ at the time of the push, and we partition the saturating pushes into three classes: (1) pushes of value $< \Delta/\beta$; (2) terminal pushes of value $\geq \Delta/\beta$; (3) nonterminal pushes of value $\geq \Delta/\beta$. The first two classes are easy to handle, whereas the number of pushes in the third class is bounded using Lemma 4.2 below. The lemma generalizes the potential function argument of [CH89, Lemma 2]. For $V' \subseteq V$ and $\gamma \geq 1$, call a push on an edge (u, v) a (V', γ)-*push* if $|\{w \in V' : d(w) = d(v)\}| \geq \gamma$ at the time of the push. A nonterminal push is just a (V, β)-push; the more general form of the lemma is needed in Sections 7 and 8. Compared with Lemma 2 of [CH89], which corresponds to the case

$\gamma = 1$, the present lemma bounds the amount of flow moved by (V', γ)-pushes by a quantity essentially inversely proportional to γ.

Lemma 4.2: For $i = 1, 2, \ldots$ and for all $V' \subseteq V$ and $\gamma \geq 1$, the total value of all (V', γ)-pushes in phase i is at most $(3n\Delta_i + 2m_i\Delta_i/\beta)|V'|/\gamma + 3\Delta_i \cdot \#relabels_i$.

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

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

Intuitively, $d'(v)$ counts the maximum number of "dense virtual distance levels" between v and t, where a vertex $v_k \in V'$ is allowed to occupy any one virtual distance level numbered at least $d(v_k)$, 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) $\forall u, v \in V : d(u) > d(v) \Rightarrow d'(u) \geq d'(v)$;

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

(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

$$\Phi = \sum_{v \in V^*,\, e^*(v) < 3\Delta_i} e^*(v) \cdot d'(v) + \sum_{v \in V^*,\, e^*(v) \geq 3\Delta_i} e^*(v) \cdot h/\gamma.$$

At the start of phase i, $\Phi \leq (3n\Delta_i + 2m_i\Delta_i/\beta)h/\gamma$ (by Fact 4), and $\Phi \geq 0$ always. Φ does not increase due to push operations (by property (2) and Fact 3), and a relabeling increases Φ by at most $3\Delta_i$ (by property (4)). It follows that the total increase in Φ during phase i is at most $3\Delta_i \cdot \#relabels_i$. Consequently, the total decrease in Φ during phase i is at most $(3n\Delta_i + 2m_i\Delta_i/\beta)h/\gamma + 3\Delta_i \cdot \#relabels_i$. Finally note that each (V', γ)-push of value c causes Φ to decrease by at least c (by property (3)). ∎

Lemma 4.3: $\#pushes = O(nm/\beta + n^2\beta + n^2 \log U)$.

Proof: Call a push *small* if its value is $< \Delta/\beta$. We prove the following claims:

(1) The total number of small saturating pushes is $O(nm/\beta + n^2 \log U)$.

(2) The total number of terminal saturating pushes is $O(n^2\beta)$.

(3) For $i = 1, 2, \ldots$, the number of nonsmall nonterminal saturating pushes in phase i is $O(nm_i/\beta + n^2 + \#relabels_i \cdot \beta)$.

(4) For $i = 1, 2, \ldots$, the number of nonsaturating pushes in phase i is $O(nm_i/\beta + n^2 + \#relabels_i)$.

Each push is counted at least once. Since $\sum_i \#relabels_i = O(n^2)$ by Lemma 3.3 and $\sum_i m_i \leq m$, the lemma follows by summing the contributions of (3) and (4) over all phases and adding those of (1) and (2). We next prove (1)–(4).

(1) Each $e \in E^*$ which is not incident on s has $ucap(e) \geq \Delta/\beta$. Hence between any two small saturating pushes on an edge $e \subset E^*$, there is a nonsaturating push on one of the edges e and $rev(e)$. The claim now follows from (4) by summation over all phases.

(2) By Fact 2, 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 $O(n^2\beta)$ pushes.

(3) Apply Lemma 4.2 with $V' = V$ and $\gamma = \beta$.

(4) Note that the value of each nonsaturating push is at least Δ and that every push is a $(V, 1)$-push and apply Lemma 4.2 as in (3), but with $\gamma = 1$. ∎

Using the definition of $T'_{cc}(n, q)$ given in the following section (for the time being, interpret $T'_{cc}(n, q)$ as T_{cc}), we can sum up the findings of this section as follows:

Theorem 1: A maximum flow in a network with n vertices, m edges and integer capacities bounded by U can be computed deterministically using $O(q)$ flow operations and $O(q + T'_{ce}(n, q))$ time, where $q = n^{3/2}m^{1/2} + n^2 \log U$.

Proof: Put $\beta = \lfloor \sqrt{m/n} \rfloor$ and combine Lemmas 4.1 and 4.3. ∎

Remark: Using the wave scaling technique of [AOT89], the value of q in Theorem 1 can be reduced to $n^{3/2}m^{1/2} + n^2(\log U)^{1/2}$.

5. Finding current edges

This section discusses the implementation of the function *ce*. We consider the following abstraction of the problem: Let $n \in \mathbb{N}$ and $V = \{1, \ldots, n\}$. The task is to maintain two functions $r : V \times V \to \{0, 1\}$ and $d : V \to \{0, \ldots, 2n - 1\}$ and n permutations $\mu_1, \ldots, \mu_n$ of V under the operations specified below. Elements of V and of $V \times V$ are called *vertices* and *edges*, respectively, and an edge $(v, w) \in V \times V$ is *eligible* if $r(v, w) = 1$ and $d(v) = d(w) + 1$. For $v \in V$, let $E(v) = \{w \in V : (v, w) \text{ is eligible}\}$.

init$(\mu'_1, \ldots, \mu'_n)$.
Precondition: $\mu'_1, \ldots, \mu'_n$ are permutations of V.
Sets $d(v) := 0$ and $\mu_v := \mu'_v$ for all $v \in V$ and $r(v, w) := 0$ for all $(v, w) \in V \times V$;

push$((v, w), b)$.
Precondition: $(v, w) \in V \times V$, $b \in \{0, 1\}$, and (v, w) is eligible.
Sets $r(w, v) := 1$ and $r(v, w) := b$.

relabel(v).
Precondition: $E(v) = \emptyset$ and $d(v) < 2n - 1$.
Executes $d(v) := d(v) + 1$.

add edge(v, w).
Precondition: $(v, w) \in V \times V$ and $d(v) \leq d(w)$.
Sets $r(v, w) := 1$.

ce(v).
Precondition: $v \in V$.
Returns $(v, \mu_v(\min\{i \in V : \mu_v(i) \in E(v)\}))$ if $E(v) \neq \emptyset$, *nil* otherwise.

The interpretation is as follows: Vertices and edges correspond to vertices and edges of G, μ_v represents the ordering of the adjacency list of v, for all $v \in V$, $r(v, w) = 1$ corresponds to (v, w) being residual, for all $(v, w) \in E^*$, *relabel*, *add edge* and *ce* correspond to the routines of the same names in the maximum flow algorithms, and *push*$(e, 0)$ and *push*$(e, 1)$ correspond to a saturating push and a nonsaturating push on e, respectively.

For $n, q \in \mathbb{N}$, denote by $T_{ce}(n, q)$ the time needed to execute any legal sequence of one *init* operation followed by q *push*, *relabel*, *add edge* and *ce* operations. Note that the symbol T_{ce} is used without arguments in a related, but different sense.

During the execution of a legal sequence of the above operations, starting with *init*, if an edge (v, w) is ineligible at some time, then it remains ineligible until the next execution of *relabel*(v) (*cf.* Fact 2). Hence we can implement *ce*(v) by letting a pointer $z[v]$ sweep over $1, \ldots, n$ until an element of $\mu_v^{-1}(E(v))$ is found, saving $z[v]$ between calls of *ce*(v), and resetting $z[v]$ to 1 in each call of *relabel*(v). Since the total number of calls of *relabel* is $O(n^2)$, it follows that $T_{ce}(n, q) = O(q + n^3)$ (another immediate bound is $T_{ce}(n, q) = O(q + nm)$, where m is the number of *add edge* operations).

We now give a faster solution for the special case in which the arguments $\mu'_1, \ldots, \mu'_n$ of *init* are all the identity permutation id_V of V. For $n, q \in \mathbb{N}$, let $T'_{ce}(n, q)$ be the quantity defined as $T_{ce}(n, q)$ for this special case. If we represent the function d not only directly, but

also through an array $D : \{0, \ldots, 2n-1\} \times V \to \{0,1\}$ such that for all $0 \le k \le 2n-1$ and all $v \in V$, $D[k,v] = 1$ if and only if $d(v) = k$, the edge $(v, z|v|)$ is eligible if and only if $r(v, z|v|) \cdot D[d(v)-1, z|v|] \neq 0$. We combine this observation with the "four Russians' trick" (see [AHU, Section 6.6]) to obtain a faster algorithm. Considering r and D as (two-dimensional) bit matrices, note that for $\mu_1 = \cdots = \mu_n = Id_V$, the search for an eligible edge with tail v is a left-to-right scan of one fixed row of r and one fixed row of D. Partitioning r and D into *blocks*, i.e., $1 \times \lfloor \log_2 n \rfloor$ submatrices, we can store the $\lfloor \log_2 n \rfloor$ bits of each block in a single RAM word, i.e., as one integer, and process the block in constant time using table look-up. This speeds up the scan by a factor of $\Theta(\log n)$ and allows q operations, starting with *init*, to be executed in $O(q + n^3/\log n)$ time. The necessary tables can be constructed in $O(n^2)$ time. We hence have

Lemma 5.1: $T^*_{cs}(n,q) = O(q + n^3/\log n)$. ∎

6. 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 $val : E_F \to \mathbb{R}$. F at all times is a directed forest, i.e., an acyclic directed graph with maximum outdegree at most one. A vertex $v \in V$ is called a *root* exactly if its outdegree in F is zero. The following operations are applied to F:

init$F$.
Sets $E_F = \emptyset$.

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 $E_F \backslash \{e\}$.

find value(e).
Precondition: $e \in E_F$.
Returns $val(e)$.

find bottleneck(v,c).
Precondition: $v \in V$ and $c \in \mathbb{R}$.
Returns the first edge e with $val(e) \le c$ on the maximal path in F starting at v, or *nil* if no such edge exists.

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.

Using the dynamic trees data structure of Sleator and Tarjan [ST85], the six operations defined above can be implemented to take $O(\log n)$ amortized time each, i.e., a sequence of q operations on F, starting with *init*F, can be executed in $O(q \log n)$ time (the *find bottleneck* operation is nonstandard, but can be implemented within this time bound).

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 *set flow* 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*);
  set flow(*e*, *cap*(*e*) − *find value*(*e*));
  cut(*e*);

The procedure *tree push* defined below works as follows: A call *tree push*(e, c) first inserts e into E_F, if it is not already in E_F, and then sends c units of flow from the tail of e along the unique path in F as far as possible without saturating any edge. If the flow does not reach a root, the first edge e' on the path with $rescap(e') \leq c$ is removed from E_F and saturated.

procedure *tree push*(*e*: *edge*; *c*: *real*);
  if $e \notin E_F$ **then** *Link*(*e*);
  $v := tail(e)$;
  $e' :=$ *find bottleneck*(v, c);
  if $e' \neq nil$ **then** *Cut*(e');
  add value(v, c);
  if $e' \neq nil$ **then** *saturate*(e');

As in Section 4, an execution of the algorithm is divided into phases parameterized by the value of a variable Δ. For $i = 1, 2, \ldots$, let Δ_i be the value of Δ in phase i. For $i = 1, 2, \ldots$, Δ_i satisfies the following requirements:

(1) $\Delta_i \leq \Delta_{i-1}/2$ (take $\Delta_0 = \infty$).
(2) At the beginning of phase i, $e^*(v) < 2\Delta_i + 2\deg_+(v)\Delta_i/\beta$ for all $v \in V^+$.
(3) At the beginning of phase i, $e^*(v) \geq \Delta_i$ for at least one vertex $v \in V^+$.

If requirement (3) cannot be satisfied for any $\Delta_i > 0$, the algorithm terminates.

The routine *select* returns a vertex $v \in V^*$ with $e^*(v) \geq \Delta$. If necessary, the current phase is first ended, and a new phase is begun.

function *select*: *vertex*;
  while $\max\{e^*(v) : v \in V^+\} < \Delta$
  do begin
    $\Delta := \min\{\Delta/2, \max(\{e^*(v) : v \in V^+\} \cup \{\beta \cdot ucap(e) : (e) \in E \backslash E^*\})\}$;
    if $\Delta = 0$ **then stop**;
    $(* \forall v \in V^+ : e^*(v) < 2\Delta; \quad \forall e \in E \backslash E^* : ucap(e) < 2\Delta/\beta *)$
    for all $e \in E \backslash E^*$ **with** $ucap(e) \geq \Delta/\beta$ **do** *add edge*(e);
  end;
  Among the vertices $v \in V^+$ with $e^*(v) \geq \Delta$, return one with minimal $d(v)$;

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

procedure *relabel*(*v*: *vertex*);
  for all $u \in V$ **with** $ce(u) = (u, v) \in E_F$ **do** *Cut*(u, v);
  $d(v) := d(v) + 1$;

Incremental strongly polynomial algorithm:
  generic initialize;
  Suitably permute the adjacency lists of G (see Section 9);
  initF; $\Delta := \infty$;
  repeat
    $v :=$ *select*;
    if $ce(v) = nil$
    then *relabel*(v)
    else *tree push*$(ce(v)$, **if** $e^*(v) \geq 2\Delta$ **then** Δ **else** $e^*(v))$;
  forever.

7. Analysis of the strongly polynomial algorithm

Again, the algorithm is easily seen to be an instance of the incremental generic algorithm. Note that F remains acyclic, as required, since E_F at all times is a subset of the set of current edges. This and the following sections investigate the running time of the algorithm. The symbols *ucap*, $\mathcal{S}$, T_{cc}, $\deg_i(v)$, m_i, and $\sharp relabels$, are used with the same meaning as in Section 4.

Define a *cycle* to be one iteration of the main loop of the algorithm. An execution of *Link*(e) and *Cut*(e) is called a *link* on e and a *cut* on e, respectively. A call of *select* will be called a *select step*, and a v-*select* if it returns the vertex v. Let $\sharp selects$ and $\sharp cuts$ denote the total number of select steps and cuts, respectively. Facts 1–4 and Lemma 4.2 still hold. In addition, we have

Fact 5: While $v \in V$ is not a root, $e^*(v)$ does not increase due to nonsaturating pushes into v.

Fact 6: At the end of a cycle containing a v-select, either v is a root, $e^*(v) = 0$, or $e^*(v) \geq \Delta$.

Fact 7: Following each cut on an edge $e \in E$ and in the same cycle, e becomes ineligible.

Lemma 7.1: The algorithm uses $O(Q)$ flow operations and $T_{cc} + O(Q)$ time, where $Q = \sharp selects \cdot \log n + m \log(n\beta)$.

Proof: In order to efficiently compute the maxima and minima needed in *select*, we maintain two heaps, the d-heap, containing all vertices $v \in V$ with $e^*(v) \geq \Delta$, ordered according to the key $d(v)$, and the e-heap, containing all vertices $v \in V$ with $e^*(v) < \Delta$, ordered according to the key $-e^*(v)$. We assume a standard heap implementation with a logarithmic time bound for each operation. In particular, a push operation, which must update at most two values stored in the heaps, can be executed in $O(\log n)$ time. Decreasing Δ is expensive, since possibly many vertices must be transferred from the e-heap to the d-heap. However, only one vertex is removed from the d-heap per *select* step, so that the total time spent in decreasing Δ is $O((\sharp selects + n) \log n)$. The operations that modify E^* can be executed in $O(m \log n)$ time. Altogether, hence, the total time spent in calls of *select* is $O((\sharp selects + m) \log n)$.

Each call of *treepush* executes $O(1)$ operations on F, and the number of *cut* operations executed in *relabel* cannot exceed the number of *link* operations executed in *treepush*. Hence the total number of operations executed on F is $O(\sharp selects)$, for a total time of $O(\sharp selects \cdot \log n)$. The remaining parts of *treepush* and *relabel* can be executed in $O(\sharp selects \cdot \log n)$ time, provided that a list of the edges in E_F entering v is maintained for each $v \in V$. Finally, $O(m \log(n\beta))$ time suffices for the initialization. ∎

Lemma 7.2: $\sharp selects = O(\sharp cuts + n^2)$.

Proof: Define a v-select to be *decreasing* if $e^*(v)$ decreases by at least Δ in the same cycle. A nondecreasing v-select is followed in the same cycle by a relabeling of v or a cut on an edge with tail v. By Lemma 3.3, it therefore suffices to count the number of decreasing select steps.

Call a vertex $v \in V^+$ *special* if $e^*(v) \geq 3\Delta$, and call a select step special if it returns a special vertex. Since no vertices become special during a phase and since $e^*(v)$ never increases in a phase while v is special (by Fact 3), the total number of special decreasing v-selects in phase i is seen by Fact 4 to be at most $2\deg_i(v)/\beta$. Summing over all vertices and all phases shows the total number of special decreasing select steps to be at most $2m/\beta$.

In order to count the remaining select steps, define a *major event* for a vertex v to be a relabeling of v, a saturating push into v, a link or a cut on an edge leaving v, the addition to E^* of an edge with head v, or program initialisation or termination. We will count the number of nonspecial decreasing v-selects in a particular period between two successive major events for v. Either v is a root throughout the period (Case 1), or v is a nonroot throughout the period (Case 2).

Case 1: At most one decreasing v-select can occur during the period.

Case 2: By Fact 5, $e^*(v)$ never increases during the period. At the time of the first nonspecial v-select in the period, $e^*(v) < 3\Delta$, and if Δ is changed during the period, $e^*(v)$ first decreases

to zero (by Fact 6). Hence there are at most 2 nonspecial decreasing v-selects during the period. Summing over all periods and all vertices, we find that the total number of nonspecial decreasing select steps is at most $2n$ plus twice the number of major events, which is $O(\sharp cuts + n^2)$. ∎

Define the *status* of an edge $e \in E$ as follows: While $e \in E \backslash E^*$, e is *absent*. For $e \in E^*$, e is *medium* if $ucap(e) \leq 20n^3\Delta$, and e is *huge* if $ucap(e) > 20n^3\Delta$.

Lemma 7.3: Let $e \in E$ be huge. Then at least one of the edges e and $rev(e)$ is never again saturated.

Proof: Applying Lemma 4.2 with $V' = V$ and $\gamma = 1$ shows the total value of all pushes in phase i to be at most $(3n^2 + 2mn + 6n^2)\Delta_i \leq 5n^3\Delta_i$, for $i = 1, 2, \ldots$, and hence the increase in $f(e)$ in phase i and all subsequent phases to be at most $10n^3\Delta_i$. ∎

Define a cut to be a *PTR event* if it happens during an execution of *relabel*, and denote by $\sharp ptr$ the total number of PTR events during the execution. PTR events were introduced in [CH89].

Lemma 7.4 ([CH89], Lemma 6): Over the whole execution, there are $O(\sharp ptr + m)$ cuts on huge edges. ∎

8. Operations on medium edges

In order to bound $\sharp cuts$, it turns out to be essential to count the number of certain pushes of value $\geq \Delta/\beta$ on medium edges. We next introduce some convenient terminology for speaking about pushes. A push over an edge (u, v) happening while $d(v) = k$ is represented by the triple $\langle u, v, k \rangle$.

Define an *event list* to be a repetition-free sequence of triples of the form $\langle u, v, k \rangle$, where $(u, v) \in E$, $0 \leq k \leq 2n - 1$, and at some time during the execution, (u, v) is medium while simultaneously $d(v) = k$. Given an event list Ψ and triples t_1 and t_2, we write $t_1 \prec_\Psi t_2$ to indicate that t_1 and t_2 both occur in Ψ, with t_1 preceding t_2.

We also need to formalize the notion that a vertex is incident with a large number of currently medium edges. For $v \in V$, denote by $\deg(v)$ the number of edges in E with head v. A phase (an integer k, respectively) is said to *hit* a vertex v if v is the head of at least $\deg(v)/\beta$ edges that are medium in that phase (throughout that part of the execution in which $d(v) = k$, respectively). For $i = 1, 2, \ldots$, denote by V_i the set of vertices hit by phase i and let $n_i = |V_i|$. A push occurring in phase i is called *regular* if it is a (V_i, β)-push and its value is at least Δ_i/β.

Each push of interest will be either regular or associated with a terminal triple in a suitably defined event list, where a triple $\langle u, v, k \rangle$ in an event list Ψ is called *terminal* (with respect to Ψ) if

$$|\{w : \langle u, v, k \rangle \prec_\Psi \langle u, w, k \rangle \text{ and } w \text{ is hit by } k\}| < \beta.$$

Hence our immediate objective is to count terminal triples and regular pushes.

Lemma 8.1: The number of terminal triples in an event list is $O(nm/\beta + n^2\beta)$.

Proof: Since an edge changes its status at most twice, Ψ contains $O(m)$ triples $\langle u, v, k \rangle$ such that (u, v) changes its status while $d(v) = k$. Let Ψ' be the set of remaining triples in Ψ.

For each $v \in V$ and $0 \leq k \leq 2n - 1$ such that v is not hit by k, Ψ' contains less than $\deg(v)/\beta$ triples of the form $\langle u, v, k \rangle$. Summing over all v and k, this yields nm/β triples. For each $u \in V$ and $0 \leq k \leq 2n - 1$, Ψ' contains less than β terminal triples of the form $\langle u, v, k \rangle$, where v is hit by k. Summing over all u and k gives $O(n^2\beta)$ triples. It is easy to see that each terminal triple in Ψ has been counted (at least once). ∎

Lemma 8.2: The total number of regular pushes is $O(nm/\beta + n^2\beta \log n)$.

Proof: One easily shows that each vertex is hit by $O(\beta \log n)$ phases. Hence $\sum_i n_i = O(n\beta \log n)$. By Lemma 4.2, applied with $V' = V_i$ and $\gamma = \beta$, the total number of regular pushes is

$$O(\sum_i (nn_i/\beta + m_i n_i/\beta^2 + \sharp relabels_i) \cdot \beta) = O(nm/\beta + n^2\beta \log n). \quad \blacksquare$$

Lemma 8.3: Over the whole execution, there are $O(nm/\beta + n^2\beta \log n + \sharp ptr)$ cuts on medium edges.

Proof: Consider a cut on a medium edge $e = (u, v)$. We can assume that the cut is not the first cut on e and that both the previous cut on e and the cut under consideration happen in an execution of *treepush* (i.e., they are not PTR events). Let $\mathcal{I}$ be the part of the execution between the end of the cycle containing the previous cut on e and the end of the cycle containing the cut under consideration. We consider two cases:

Case 1: $rescap(e) < \Delta/\beta$ throughout $\mathcal{I}$. In this case each of the ≥ 1 cuts on $rev(e)$ during $\mathcal{I}$ is a PTR event. Hence Case 1 contributes $O(\sharp ptr)$ cuts.

Case 2: $rescap(e) \geq \Delta/\beta$ at some time during $\mathcal{I}$. Associate with the cut one distinguished push over e during $\mathcal{I}$ that changes $rescap(e)$ from $\geq \Delta/\beta$ to $< \Delta/\beta$ and note that the value of this push is at least Δ/β. Furthermore, if $d(v) = k$ at the time of the distinguished push, associate with the push the triple $\langle u, v, k\rangle$ and append (u, v, k), at the time of the push, to an initially empty sequence Ψ.

The final value of Ψ is an event list, and each push associated with a nonterminal triple in Ψ is regular. It now follows from Lemmas 8.1 and 8.2 that Case 2 contributes $O(nm/\beta + n^2\beta \log n)$ cuts. $\blacksquare$

Lemma 8.4: The algorithm uses $O(Q)$ flow operations and $T_{rf} + O(Q)$ time, where $Q = n^{3/2}m^{1/2}(\log n)^{3/2} + \sharp ptr \cdot \log n$.

Proof: Put $\beta = 1 + \lfloor m^{1/2}n^{-1/2}(\log n)^{-1/2} \rfloor$ and combine Lemmas 7.1, 7.2, 7.4 and 8.3. $\blacksquare$

9. PTR events

The number of PTR events may depend on the ordering of the adjacency lists of G, which defines ce. We need some technical definitions to discuss this dependence.

For every finite set A, denote by Π_A the set of all permutations of A, i.e., of all bijections $\pi : \{1, \ldots, |A|\} \to A$. For every $A' \subseteq A$ and every $\xi \subset \Pi_{A'}$ and $\sigma \subset \Pi_A$, denote by $\lambda(\xi, \sigma)$ the length of a longest (not necessarily contiguous) ascending subsequence of the sequence $\sigma^{-1}(\xi(1)), \ldots, \sigma^{-1}(\xi(|A'|))$ or, equivalently, the length of a longest (not necessarily contiguous) common subsequence of the sequences $\xi(1), \ldots, \xi(|A'|)$ and $\sigma(1), \ldots, \sigma(|A|)$. Finally, for any set $\{\xi_1, \ldots, \xi_n\}$ of permutations of subsets of a finite set A, let $\Lambda(\xi_1, \ldots, \xi_n) = \max_{\sigma \in \Pi_A} \sum_{i=1}^{n} \lambda(\xi_i, \sigma)$.

Let $V = \{v_1, \ldots, v_n\}$ and for $i = 1, \ldots, n$, let $\Gamma_i = \{w \in V : (v_i, w) \subset E\}$ and $d_i = |\Gamma_i|$. For $i = 1, \ldots, n$, the ordering of the adjacency list of v_i may be viewed as a permutation ξ_i of Γ_i, i.e., $(v_i, \xi_i(j))$ is the jth edge in the adjacency list of v_i, for $j = 1, \ldots, d_i$. The following fact was essentially proved in [CH89] (Lemma 9 and Claim following Lemma 11):

Lemma 9.1: If the adjacency list of v_i is ordered according to $\xi_i \in \Pi_{\Gamma_i}$, for $i = 1, \ldots, n$, then $\sharp ptr \leq 2n \cdot \Lambda(\xi_1, \ldots, \xi_n)$. $\blacksquare$

The fact below was also essentially proved in [CH89] and expressed there as Lemma 10 (put $\beta = \sqrt{m/n}$).

Lemma 9.2: Suppose that ξ_i is drawn randomly from the uniform distribution over Π_{Γ_i}, for $i = 1, \ldots, n$, and that $\xi_1, \ldots, \xi_n$ are independent. Then for any $r \geq \sqrt{nm} + n \log n$, $\Lambda(\xi_1, \ldots, \xi_n) = O(r)$ with probability at least $1 - 2^{-r}$. $\blacksquare$

Combining Lemmas 8.4, 9.1 and 9.2, we obtain

Theorem 2: For any constant $\alpha > 0$, a maximum flow in a network with n vertices and m edges can be computed using $O(Q)$ flow operations and $O(Q + T_{cc}(n, Q/\log n))$ time with probability at least $1 - 2^{-\alpha\sqrt{n \log n}}$, where $Q = O(n^{3/2} m^{1/2} (\log n)^{3/2} + n^2 (\log n)^2)$. ∎

Alon has given a deterministic construction of pseudo-random permutations with properties similar to those exploited above.

Lemma 9.3 ([Al89], Theorem 2): For every two integers n and h with $n \geq h \geq 1$ and every set V with $|V| = h$, n permutations $\xi_1, \ldots, \xi_n$ of V with $\Lambda(\xi_1, \ldots, \xi_n) = O(nh^{2/3})$ can be constructed in $O(nh)$ time. ∎

Theorem 3: A maximum flow in a network with n vertices can be computed deterministically using $O(Q)$ flow operations and $O(Q + T_{cc}(n, Q/\log n))$ time, where $Q = O(n^{8/3} \log n)$. ∎

The fast solution to the current-edge problem described in Section 5 assumes identical orderings of all adjacency lists. As we saw above, however, it is essential to order different adjacency lists differently. Let $B = \{b_1, \ldots, b_{n/x}\}$ be a partition of V into *blocks* $b_1, \ldots, b_{n/x}$ of size $x = \lfloor \log_2 n \rfloor$ each and corresponding in the obvious way to the blocks defined in Section 5. Different permutations of the blocks in different adjacency lists is easily accommodated, but the association between vertices and blocks is fixed by the interpretation of D and must be the same for all adjacency lists. Hence not all permutations of V represent possible adjacency list orderings, and therefore Alon's scheme (Lemma 9.3) cannot be used without modification. Our solution is to apply the scheme to the ordering of blocks instead of to the ordering of vertices.

For every *block permutation* $\xi \in \Pi_B$, define the induced *full permutation* as the permutation $\hat{\xi} \in \Pi_V$ 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 $w \in b_j$, $\hat{\xi}^{-1}(v) < \hat{\xi}^{-1}(w) \iff (\xi^{-1}(b_i) < \xi^{-1}(b_j)$ or $(i = j$ and $v < w))$).

Lemma 9.4: For any $\xi_1, \ldots, \xi_n \in \Pi_B$, $\Lambda(\hat{\xi}_1, \ldots, \hat{\xi}_n) \leq x \cdot \Lambda(\xi_1, \ldots, \xi_n)$.

Proof: Fix $\sigma \in \Pi_V$ arbitrarily and let $R \subseteq \Pi_B$ be the set of those block permutations ρ that can be obtained as follows: For $i = 1, \ldots, n/x$, select a *representative* $r_i \in b_i$ from b_i, and then arrange the blocks in the order in which their representatives occur in σ (i.e., for $1 \leq i, j \leq n/x$, $\rho^{-1}(b_i) < \rho^{-1}(b_j) \iff \sigma^{-1}(r_i) < \sigma^{-1}(r_j)$). We call $r_1, \ldots, r_{n/x}$ the *defining vertices* of ρ. Now, for any block permutation $\xi \in \Pi_B$,

$$\sum_{\rho \in R} \lambda(\xi, \rho) \geq \frac{|R|}{x} \lambda(\hat{\xi}, \sigma).$$

To see this, note that each element of a fixed longest common subsequence of $\hat{\xi}(1), \ldots, \hat{\xi}(n)$ and $\sigma(1), \ldots, \sigma(n)$ contributes 1 to $\lambda(\xi, \rho)$ if it is a defining vertex of ρ, and that each $v \in V$ is a defining vertex of exactly $|R|/x$ permutations $\rho \in R$. Summing the above inequality for ξ equal to $\xi_1, \ldots, \xi_n$ produces

$$\sum_{i=1}^{n} \lambda(\xi_i, \sigma) \leq \frac{x}{|R|} \sum_{i=1}^{n} \sum_{\rho \in R} \lambda(\xi_i, \rho) = \frac{x}{|R|} \sum_{\rho \in R} \sum_{i=1}^{n} \lambda(\xi_i, \rho)$$

$$\leq \frac{x}{|R|} \sum_{\rho \in R} \Lambda(\xi_1, \ldots, \xi_n) = x \cdot \Lambda(\xi_1, \ldots, \xi_n). \quad ∎$$

By Lemma 9.3, n block permutations $\xi_1, \ldots, \xi_n \in \Pi_B$ with $\Lambda(\xi_1, \ldots, \xi_n) = O(n(n/\log n)^{2/3})$ can be constructed in $O(n^2/\log n)$ time. By Lemmas 9.1 and 9.4, if the n adjacency lists of G are ordered according to $\hat{\xi}_1, \ldots, \hat{\xi}_n$, then $fptr \leq n^{8/3} (\log n)^{1/3}$. As argued above, Lemma 5.1 can be generalized to the case where the arguments $\mu'_1, \ldots, \mu'_n$ of *init* are arbitrary full permutations induced by block permutations. Our main result follows by an appeal to Lemma 8.4.

Theorem 4: 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. ∎

10. Additional results

The analysis of the PLED algorithm [CH89] can be improved using the approach of Section 8. This yields the following result, which was first obtained by Tarjan [Ta89] using a different method.

Theorem 5: For any constant $\alpha > 0$, the PLED algorithm finds a maximum flow in $O(nm + n^2(\log n)^2)$ time with probability at least $1 - n^{-\alpha n^2}$.

Since our solution to the bottleneck current-edge problem trivially parallelizes on most parallel machines, it is possible to crank out a variety of parallel algorithms for the maximum-flow problem that are optimal, as measured by the best currently known sequential algorithms. We mention just one example. As is to be expected because of the P-completeness of the maximum-flow problem [GSS82], the algorithms are optimal only for relatively long execution times. No optimal parallel algorithm for the maximum-flow problem (using $\omega(1)$ processors) was previously known.

Theorem 6: For $p \leq n^{3/3}(\log n)^{-7/3}$, 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 p processors interconnected to form a complete binary tree.

Late note: Very recently we have discovered alternative algorithms that allow the value of Q in Theorem 2 to be reduced, with a slightly weaker probability bound, to $Q = O(n^{3/2}m^{1/2}\log\alpha + \alpha^2(\log n)^2)$ in the general case and to $Q = O((n^{3/2}m^{1/2} + n^2\log n)\log(2 + (n/m)\log U))$ in the case of integer capacities bounded by U.

References

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

[AO87] R. K. Ahuja and J. B. Orlin, A Fast and Simple Algorithm for the Maximum Flow Problem, Sloan W.P. No. 1905–87 (revised), MIT, October 1988.

[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.

[Al89] N. Alon, Generating Pseudo-Random Permutations and Maximum Flow Algorithms, manuscript, December 1989.

[CH89] J. Cheriyan and T. Hagerup, A Randomized Maximum-Flow Algorithm, Proceedings, 30th Annual Symposium on Foundations of Computer Science (1989), pp. 118–123.

[GT88] A. V. Goldberg and R. E. Tarjan, A New Approach to the Maximum-Flow Problem, *J. ACM* 35 (1988), pp. 921–940.

[GSS82] L. M. Goldschlager, R. A. Shaw and J. Staples, The Maximum Flow Problem is Log Space Complete for P, *Theor. Comp. Sci.* 21 (1982), pp. 105–111.

[ST85] D. D. Sleator and R. E. Tarjan, Self-Adjusting Binary Search Trees, *J. ACM* 32 (1985), pp. 652–686.

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