Approximate max flow on small depth networks

Edith Cohen
AT&T Bell Laboratories
Murray Hill, NJ 07974

Revised: July 1993

Abstract

We consider the maximum flow problem on directed acyclic networks with m edges and *depth* r (length of the longest s-t path). Our main result is a new deterministic algorithm for solving the relaxed problem of computing an s-t flow of value at least $(1-\epsilon)$ of the maximum flow. For instances where r and ϵ^{-1} are small (i.e., $O(\text{polylog}(m))$), this algorithm is in $\mathcal{NC}$ and uses only $O(m)$ processors, which is a significant improvement over existing parallel algorithms. As one consequence, we obtain an $\mathcal{NC}$ $O(m)$ processor algorithm to find a bipartite matching of cardinality $(1-\epsilon)$ of the maximum (for $\epsilon^{-1} = O(\text{polylog}(m))$). We use a novel approach based on path-counts to compute blocking flows in parallel. This approach produces fractional flow even when capacities are integral. To encounter that, we provide a rounding algorithm that is of independent interest. In polylogarithmic time using $O(m)$ processors, the algorithm rounds any fractional flow on a network with integral capacities to an integral flow. The rounding technique extends to networks with costs.

1. Introduction

A flow network is modeled by a directed acyclic graph $G = (V, E)$ with capacities $u_e \geq 0$ $(e \in E)$ associated with the edges, a designated *source* vertex $s \in V$, and a designated *sink* vertex $t \in V$. We use the parameter r to denote the *depth* of the network, which we define to be the length of the longest s-t path in G. For a vertex $v \in V$, we denote by $\text{in}(v)$ (resp., $\text{out}(v)$) the set of edges directed into (resp., out of) v. An *s-t flow* f on G is an assignment of flow values f_e $(e \in E)$ to edges such that the capacity constraints are not violated (for all $e \in E$, $0 \leq f_e \leq u_e$), and flow is conserved at every vertex other than the source or the sink, that is, for all $v \in V \setminus \{s,t\}$, $\sum_{e \in \text{in}(v)} f_e = \sum_{e \in \text{out}(v)} f_e$. The *value* of an s-t flow is the amount of excess flow at t (i.e., $\sum_{e \in \text{in}(t)} f_e - \sum_{e \in \text{out}(t)} f_e$). A maximum flow (max-flow) is an s-t flow of maximum value. We use M^* to denote the value of the max-flow.

Finding a maximum flow is one of the classic problems in combinatorial optimization. The fastest known sequential algorithms for the problem have time bounds close to $O(mn)$ (see [1, 2, 13]). The known parallel algorithms for the problem are less satisfactory. The maximum flow problem with general capacities is known to be $\mathcal{P}$-complete [16] and hence not likely to have an $\mathcal{NC}$ algorithm. The fastest known deterministic parallel algorithm solves the problem (on a CRCW PRAM) in $O(n^2 \log n)$ time using $O(n)$ processors (an algorithm by Shiloach and Vishkin [25] uses $O(n^2)$ space and a later algorithm by Goldberg and Tarjan [13] uses $O(m)$ space). Ahuja and Orlin [1] presented an $O(n^2 \log U \log(m/n))$ time $O(m/n)$ processor EREW PRAM algorithm, where U is the maximum capacity. When the capacities are polynomial in m, the problem seems easier: It is not known to be $\mathcal{P}$-complete, and furthermore, it is in $\mathcal{RNC}$ by a standard logspace reduction to the maximum cardinality bipartite matching problem. The bipartite matching problem was first shown to be in $\mathcal{RNC}$ by Karp, Upfal, and Wigderson [20], who gave an algorithm that uses $O(n^{6.5})$ processors. Later algorithms (see [10, 23]) use fewer processors, where the best current bound is $O(nM(n))$ by Galil and Pan [10]. When capacities are in $\{0, 1\}$, the maximum flow problem can be solved within the same bounds as maximum cardinality bipartite matching. If the capacities are polynomial in m, an additional factor of $O(m)$ is added to the processor bound. (When capacities are general a factor of $\log U$ is added to the running time, and hence the algorithm is not in $\mathcal{RNC}$.) To summarize, we note that max-flow with polynomial capacities can be solved sequentially using $\tilde{O}(mn)$ work, [1] in parallel in $\tilde{O}(n^2)$ time using $\tilde{O}(n^3)$ work, and in $\mathcal{RNC}$ using expected $\tilde{O}(mnM(n))$ work.

We present a max-flow algorithm with the following bounds:

Theorem 1.1 *A flow of value M such that*

$$M \geq (1 - \epsilon)M^*$$

can be computed deterministically on an EREW PRAM in

i. $O(r^3 \epsilon^{-3} \log^2 m \log(m\epsilon^{-1}))$ *time using* $O(m/r)$ *processors and* $O(m)$ *space, or*

ii. $O(r^2 \epsilon^{-2} \log^3 m \log(r\epsilon^{-1}) \log(m\epsilon^{-1}))$ *time using* $M(n)$ *processors and* $O(M(n))$ *space.*

Observe that when the depth r and ϵ^{-1} are small (e.g., $O(\mathrm{polylog}(m))$), the resource bounds obtained here are significantly better than known bounds given for general exact maximum flow algorithms.

Our max-flow algorithm consists of iterating calls to a subproblem of finding approximate *blocking flows*, and is based on a novel parallel blocking flow algorithm. A blocking flow is an s-t flow which can not be augmented without pushing flow "backward" through at least one edge. Dinic's algorithm [4] introduced a scheme to solve the max-flow problem by reducing it to blocking flow computations. Dinic's algorithm computes a max-flow in $O(n)$ phases, where each phase amounts to a blocking flow computation on a directed acyclic layered network. The depth of the layered network increases at each phase. The fastest known

[1] We use the notation $\tilde{O}(f) \equiv O(f \,\mathrm{polylog}\, f \,\mathrm{polylog}\, n)$.

sequential blocking flow algorithm is due to Goldberg and Tarjan [15] and has an almost optimal time bound of $O(m \log(n^2/m))$. The fastest known (polynomial-work) deterministic parallel algorithm for computing blocking flows was given by Goldberg and Tarjan [14] and runs in $O(n \log n)$ time, using $O(m)$ processors. When capacities are polynomial in m, the problem is in $\mathcal{RNC}$ since this is true for the more general max-flow problem. Unlike the more general $\mathcal{P}$-complete max-flow problem, there is no "hard evidence" that the blocking flow problem is not in $\mathcal{NC}$.

We give an overview of the ideas used and the structure of the paper.

In Section 2 we review commonly used terminology for network flow algorithms, sketch Dinic's maximum flow algorithm, and discuss the blocking flow problem.

In Section 3 we examine the use of Dinic's algorithm to obtain near-maximum flows. We show that in order to obtain a flow of value at least $(1-\epsilon)M^*$, it suffices to run Dinic's algorithm for at most $O(r\epsilon^{-1})$ phases (until the length of the layered networks exceeds $O(r\epsilon^{-1})$). Furthermore, we show that when only a near-maximum flow is desired, it is not necessary to find an exact blocking flow at each phase. We define a notion of an approximate blocking flow and prove that we can use a relaxed variant of Dinic's algorithm where exact blocking flow computations are replaced by approximate ones.

In Section 3 we shall see that the blocking flow instances that arise when running the approximate version of Dinic's algorithm involve networks of depth at most $O(r\epsilon^{-1})$. This fact does not benefit us when considering sequential algorithms since blocking flows can be found almost optimally regardless of the depth of the network [15]. We can gain, however, in the parallel case. The worst case behavior of the $O(n \log n)$ time Goldberg Tarjan [14] parallel blocking flow algorithm does not improve when instances are limited to networks of small depth. In Section 4 we present a novel parallel approximate blocking flow algorithm that is significantly more efficient on small depth networks than existing exact blocking flow algorithms. The algorithm uses $O(m)$ processors and is in $\mathcal{NC}$ when $r = O(\text{polylog}\, m)$. If the capacities are polynomial in m, the algorithm computes an exact blocking flow within the same resource bounds.

Our parallel blocking flow algorithm is based on a notion of *path counts* of a network. We define the path count of an acyclic network to be the sum over all s-t paths of the product of all edge capacities along the path. The path count of an edge $e \in E$ is the same sum restricted to paths that contain e. Observe that when the network has unit capacities, the path count reduces to the number of different s-t paths. The algorithm is fairly simple and we sketch it here. We make use of the property that path counts of edges are conserved at all vertices other than the source and the sink. That is, for each vertex $v \in V \setminus \{s,t\}$, the sum of path counts of edges entering v equals the sum of path counts of edges exiting v. Hence, an assignment of flow values to edges where the values are proportional (with a small enough proportion factor) to the respective path counts constitutes a legal flow. The blocking flow algorithm repeatedly computes the path counts at edges, finds the largest proportional flow augmentation that does not violate the current capacities, and updates the current flow. The following iteration is applied to the network with the recent augmentation being subtracted from the capacities. The algorithm terminates when the current flow is approximately blocking. We will show that the path counts computations, and hence each iteration, can be implemented efficiently in parallel.

The crucial property of the algorithm is that the number of iterations is small (namely, $\tilde{O}(r)$). We give some intuition why this is indeed the case. Consider, for simplicity, a network with unit capacities. A maximal independent set of s-t paths corresponds to a blocking flow. A general approach to find, in parallel, a maximal independent set in a graph is to use iterations, where in each iteration we find some independent set. In the following iteration this independent set and its neighbors are removed from the graph. To obtain a small number of iterations, we need to use independent sets with large neighborhoods. In the blocking flow setting, such an independent set corresponds to a flow augmentation that significantly reduces the path count of the network. The flow augmentation used by the algorithm achieves that by assigning more flow to edges with a large path count (have many s-t paths going through them).

The blocking flow algorithm of Section 4 conceivably terminates with a fractional flow, even when capacities are integral. In Section 5 we describe an $\mathcal{NC}$ "rounding" algorithm that uses $O(m)$ processors. For a given fractional flow of value M on any network with integral capacities, the algorithm computes an integral flow of value at least $\lceil M - 1/\operatorname{poly}(n) \rceil$. We also describe how to obtain an integral blocking flow by combining the fractional approximate-blocking flow algorithm of Section 4 with the rounding technique, and how to round a fractional circulation in a network with costs, with at most a small increase in cost. Our rounding algorithm generalizes a similar algorithm for bipartite matching networks given by Goldberg, Plotkin, Shmoys, and Tardos [11].

In Section 6 we discuss parallel exact maximum flow algorithms for zero-one and unit networks. Zero-one networks are networks where all capacities are in $\{0, 1\}$. A unit network is a zero-one network where each vertex has either at most one incident outgoing edge or at most one incident ingoing edge. We give algorithms that use balancing techniques similar to the ones used by Goldberg, Plotkin, and Vaidya in [12], and combine the blocking flow algorithm of Section 4 and a parallel version of the Ford-Fulkerson algorithm [7]. We obtain sublinear parallel running times that are comparable to the running times given in [12].

Section 7 is concerned with the maximum cardinality bipartite matching problem. By a standard reduction, an instance of the problem can be reduced to an integral max-flow problem on a zero-one network of depth $r = 3$. The corresponding max-flow instance is such that an integral flow of some value M yields a matching of cardinality M in the original problem. Hence, the results of previous sections imply an $O(\log^3 m)$ time $O(m)$ processor deterministic parallel algorithm for finding a bipartite matching of cardinality at least $(1-\epsilon)$ of the cardinality of the maximum matching when $\epsilon^{-1} = O(1)$. Note that the algorithm remains in $\mathcal{NC}$ and uses $O(m)$ processors even when $\epsilon^{-1} = O(\operatorname{polylog} m)$.

2. Preliminaries

The parallel running times and processor utilization mentioned in this paper are for the EREW PRAM model. The routines used are limited to standard ones (e.g., list rankings, BFS, shortest path computations, finding Eulerian partitions) and hence running times on other models can be deduced easily (see, e.g., the survey by Karp and Ramachandran [19]). We use $M(n)$ to denote the best currently-known upper bound on the number of operations needed to multiply two $n \times n$ matrices. It is known (see Coppersmith and Winograd [3]) that $M(n) = o(n^{2.376})$. The quantity $M(n)$ appears in some of our processor bounds for parallel algorithms, by relying on a result of Pan and Reif [24] who had shown that two $n \times n$ matrices can be multiplied in $O(\log n)$ time using $M(n)$ processors, and the fact that the best known processor bounds for $\mathcal{NC}$ computation of transitive closure, directed s-t path, and BFS tree are obtained using matrix multiplication.

We review some standard terminology associated with maximum flow algorithm and sketch Dinic's algorithm. For further details see [27]. A flow in a network is *blocking* if every path from s to t contains a saturated edge. Consider a current flow f in G and the corresponding residual graph R. The *level* of a vertex $v \in V$ is the length of the shortest path from s to v in R. The *level graph* L for f is the subgraph of R that contains only vertices reachable from s and only edges $e = (u, v)$ such that $\text{level}(v) = \text{level}(u) + 1$. The construction of the level graph for a given flow f can be done in $O(m)$ time using breadth first search. In parallel, the construction can be done in $O(r \log m)$ time using $O(m/r)$ processors, where r is the level of t (using a straightforward parallel implementation of the breadth first search algorithm).

Dinic's maximum flow algorithm [4] maintains a current flow f, initialized as the zero flow, and repeats the following until t is not reachable from s in the residual graph for f.

i. Find a blocking flow f' on the level graph for the current flow f.

ii. Replace $f \leftarrow f + f'$.

Dinic showed that the level of t strictly increases in each phase and hence the algorithm terminates after at most $n - 1$ phases.

The complexity of each iteration of Dinic's algorithm is dominated by a blocking flow computation on a layered network. It is not known whether the blocking flow problem on networks of arbitrary depth is either $\mathcal{P}$-complete or is in $\mathcal{NC}$. We sketch an $\mathcal{NC}$ but very inefficient algorithm for finding a blocking flow in networks of a fixed depth r. The algorithm is based on a similar algorithm of Fisher, Goldberg, and Plotkin [6] which was used to show that approximate matching is in $\mathcal{NC}$. In Section 4 we present a much more efficient algorithm that uses $O(m/r)$ processors and runs in $\tilde{O}(r^2)$ time (alternately, $M(n)$ processors and $\tilde{O}(r)$ time). Note that the latter algorithm is in $\mathcal{NC}$ when $r = O(\text{polylog}\, m)$.

Consider a network of depth r with $\{0, 1\}$ capacities. A blocking flow can be computed in polylogarithmic time using $n^{O(r)}$ processors as follows. The idea, roughly, is to consider all $O(n^r)$ s-t paths and construct a dependency graph such that every vertex corresponds to an s-t path and two vertices are adjacent if and only if the two corresponding paths share at least one edge. A maximal independent set in the dependency graph corresponds to a set of paths that when saturated comprise a blocking flow. The algorithm computes a maximal independent set (can be done within the stated resource bounds) and obtains a blocking flow. If capacities are integral and polynomial in m, we can use the previous method and replace edge of capacity u_e by u_e parallel edges. Note that the algorithm requires $n^{O(r)}$ processors.

When capacities are integral but possibly large, a blocking flow can be obtained by iterating the above algorithm on instances with polynomial capacities $\log(U_f/m)$ times, where U_f is the capacity of the fattest path. In each iteration, we round down the capacities to units of $\lceil U_f/m \rceil$ and truncate large capacities to $m^2\lceil U_f/m \rceil$. As a result, we obtain an instance of the blocking flow problem where all capacities are in the range $\{1..m^2\}$. In the following iteration, the flow values obtained are subtracted from the capacities, and the fattest path with respect to the updated capacities has capacity less than $\lceil U_f/m \rceil$.

3. Obtaining a near-maximum flow

We examine applying variants of Dinic's algorithm to solve the relaxed goal of obtaining a near-maximum flow, when (i) the networks have small depth or (ii) only approximately-blocking flow routines are available.

3.1. Small depth networks

Consider a network G and an s-t flow f. The following proposition states a relation between the value of f and M^* in terms of the depth r and the length of the minimum augmenting path in the residual graph induced by f.

Proposition 3.1 *Consider a network G as above, and a flow f of value M where the shortest augmenting path in the residual graph induced by f is of length $\ell \geq r+1$. Under these conditions*

$$M^* \leq (\frac{\ell + r}{\ell - r})M \ .$$

Proof: Without loss of generality, we assume integral flow values and capacities. Consider an integral flow f of value M. Denote by f^* a maximum integral flow with value M^*. There exists a set P of $M^* - M$ augmenting paths of capacity 1 from s to t on the residual graph that (i) augments the flow from f to f^* and (ii) does not use any edge in more than one direction. (Obviously, all paths in P are of length $\geq \ell$.) Note that $M^* = M + |P|$.

For the analysis, we assign vertices to layers labeled $\{0, \ldots, r\}$ where a vertex $v \in V$ is assigned according to the length of the longest path in G from s to v. It is clear that all edges are directed from lower levels to strictly higher levels. We account for a unit flow in an arc ("weight" of the arc) as follows: if it is from layer i to layer j, account for it as $|j-i|$. The total weight of a flow on G equals r times the value of the flow.

Consider a blocking flow with no augmenting path of weight less than ℓ (note that weight ℓ suffices for the result of the proposition, length ℓ is a stronger requirement). For an augmenting path $p \in P$, denote by a_p the weight from the forward edges, and by b_p the weight from the backward edges. Obviously $a_p - b_p = r$, $a_p + b_p \geq \ell$. Note that since $\ell \geq r+1$, we have $b_p \geq 1$.

Intuitively, view a path from P as "exchanging" weight b_p out of the old flow with a larger

weight a_p added toward achieving the optimal flow.
Formally: Since no two paths from P use an edge in opposite directions we have

$$\sum_{p \in P} b_p \leq rM \ .$$

From the inequalities $a_p - r = b_p$ and $a_p + b_p \geq \ell$ it follows that for all $p \in P$,

$$b_p \geq (\ell - r)/2 \ .$$

Combining the two we have

$$|P|(\ell - r)/2 \leq \sum_{p \in P} b_p \leq rM \ .$$

Hence,

$$|P| \leq \frac{2r}{\ell - r} M \ .$$

∎

Remark 3.2 Proposition 3.1 is related to a result of Even and Tarjan [5] for zero-one networks which states a relation between the length of the shortest augmenting paths in a residual graph and the value of the remaining flow. It is also related to a result by Fisher, Goldberg, and Plotkin [6] who showed that if a matching in a graph is such that there is no augmenting path of length smaller than ℓ then the cardinality of the current matching is at least $(1 - 1/\ell)$ of the maximum cardinality matching. Since a bipartite matching can be expressed as an instance of maximum flow on a directed acyclic network of depth 3, Proposition 3.1 generalizes the result of [6].

Corollary 3.3 *The value of any blocking flow on G is at least $M^*/(2r)$.*

Proof: Immediate from the fact that for a blocking flow, $\ell \geq r + 1$. ∎

Corollary 3.4 *Consider a network G as above. An s-t flow of value at least $(1 - \epsilon)M^*$ can be computed in $O(r\epsilon^{-1} m \log(n^2/m))$ time. Note that when r and ϵ are fixed this bound is almost linear.*

Proof: The stated time bound is obtained by running Dinic's algorithm for $O(r\epsilon^{-1})$ phases. Each phase amounts to a blocking flow computation which can be performed in $O(m \log(n^2/m))$ time by a result of Goldberg and Tarjan [15]. ∎

3.2. Using approximately-blocking flows

We introduce a notion of an approximate blocking flow, and a modified version of Dinic's algorithm which uses an approximate blocking flow in each iteration.

Definition 3.5 Consider a max flow instance as above. An ϵ-*blocking flow* is a flow for which all non saturated s-t paths contain an edge of capacity not greater than ϵ. A 0-blocking flow is a blocking flow.

Proposition 3.6 *[Modified Dinic Algorithm] Consider a modification of Dinic algorithm where in each phase we compute an ϵ-blocking flow on the current layered graph. In the following phase we construct a layered graph by treating edges of capacity no larger than ϵ as having capacity* 0*.*

i. At the termination of the kth phase, all augmenting paths of length at most k in the residual graph contain an edge of capacity no larger than ϵ.

ii. The depth of the layered graph strictly increases from phase to phase.

Proof: Similar to the correctness proof of Dinic's algorithm [4]. ∎

Corollary 3.7 *Consider a max flow instance as above. By running the modified Dinic algorithm until the depth of the layered network exceeds ℓ (this happens within ℓ phases) we can compute a flow of value M such that*

$$M + m\epsilon \geq (\frac{\ell - r}{\ell + r})M^* \ ,$$

where M^* is the value of the maximum flow.

Proof: This follows from Proposition 3.1 and the fact that an ϵ-optimal blocking flow can be augmented to a blocking flow using at most $O(m\epsilon)$ units of flow. ∎

In the following section we present a deterministic parallel algorithm for computing an ϵ-blocking flow. The algorithm is particularly efficient when the depth of the network is small. By combining it with Corollary 3.7 we obtain the bounds stated in Theorem 1.1.

4. Parallel ϵ-blocking flow algorithm

We consider the blocking flow problem on a directed acyclic graph $G = (V, E)$ with capacities $u_e \geq 0$ ($e \in E$), a designated source vertex $s \in V$, and a designated sink $t \in V$. The parameter r denotes the depth of the network.

We present an algorithm for computing an ϵ-blocking flow in G. We first introduce the concept of path counts.

4.1. The path counts of a network

Definition 4.1 Suppose that G is a network as above.

i. Denote by $\mathcal{P}$ the set of all s-t paths in G. Note that $|\mathcal{P}| \leq n^r$. Denote by $\mathcal{P}_{uw}$ the set of all paths from $u \in V$ to $w \in V$.

ii. The *path count* χ of the network is

$$\chi = \sum_{p \in \mathcal{P}} \prod_{e \in p} u_e \ .$$

When $u_e \in \{0, 1\}$ $(e \in E)$, then $\chi = |\mathcal{P}|$.

iii. The *path count of an edge* $e \in E$ is

$$\chi_e = \sum_{\{p \in \mathcal{P} | e \in p\}} \prod_{e' \in p} u_{e'} \ .$$

iv. The *path count from* $u \in V$ *to* $w \in V$ is

$$\chi^{uw} = \sum_{p \in \mathcal{P}_{uw}} \prod_{e \in p} u_e \ .$$

The algorithm we introduce here relies on the property that the path counts of edges obey "conservation constraints" at vertices, and hence, so does a flow function proportional to the edges' path counts.

Proposition 4.2 *For all* $v \in V \setminus \{s, t\}$,

$$\sum_{e \in \mathrm{in}(v)} \chi_e = \sum_{e \in \mathrm{out}(v)} \chi_e \ .$$

Proof: Consider a vertex u. Both quantities above equal the path count of u,

$$\chi_u = \chi^{su} \chi^{ut} = \sum_{\{p \in \mathcal{P} | u \in p\}} \prod_{e \in p} u_e \ .$$

∎

Complexity of computing the path counts. We suggest two algorithms to compute χ and χ_e for all $e \in E$. This is done by first computing for each vertex $v \in V$, the quantities χ^{sv} and χ^{vt}. It is easy to see that the path count χ_e of an edge $e \in E$, where $e = (u, v)$, equals $\chi_e = u_e \chi^{su} \chi^{vt}$.

The quantities χ^{su} and χ^{ut} can be computed as follows:

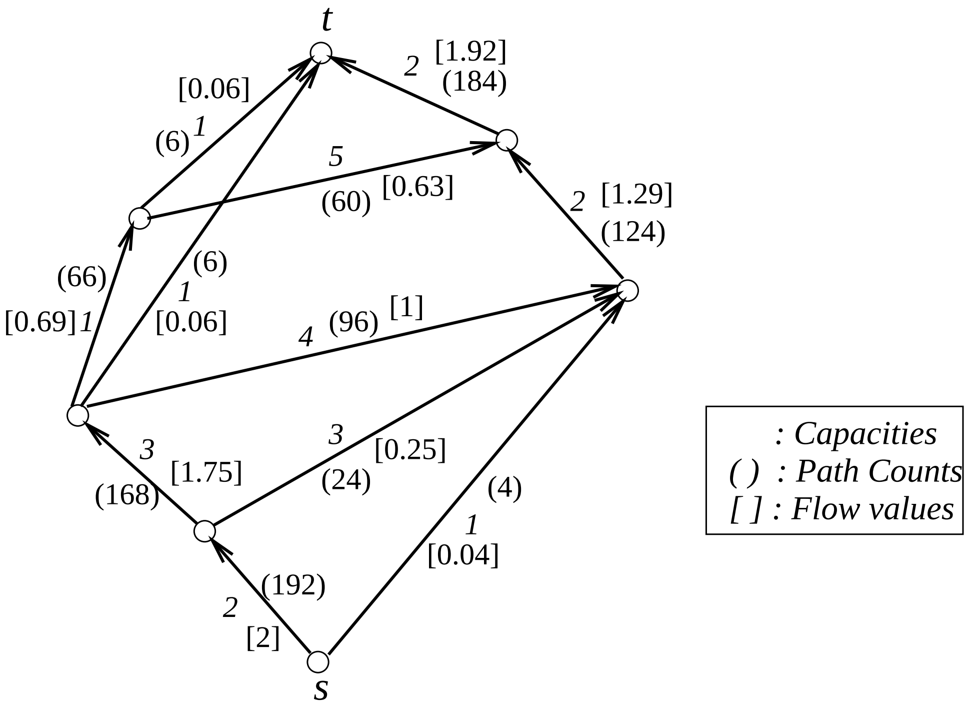

Figure 1: Example of a network with path-counts and corresponding flow values

i. We use the relations $\chi^{ss} = 1$, $\chi^{tt} = 1$ and for $v \in V$, $\chi^{sv} = \sum_{e=(w,v)\in \text{in}(v)} \chi^{sw} u_e$ and $\chi^{vt} = \sum_{e=(v,w)\in \text{out}(v)} \chi^{wt} u_e$. We do one forward pass starting at s to compute the quantities χ^{sv} $(v \in V)$, and one pass backward starting at t to compute χ^{vt} $(v \in V)$. Each pass consists of phases, where in the ith phase $(0 \le i \le r)$ we scan all vertices such that i is the size of the maximum-size path from s to the vertex (in a backward pass, from the vertex to t). Note that we scan a vertex only if all of its predecessors were scanned previously. This can be done in $O(r \log m)$ time using $O(m/r)$ processors on an EREW PRAM.

ii. By using $O(\log r)$ matrix multiplications of $n \times n$ matrices. Pan and Reif [24] have shown that multiplication of two $n \times n$ matrices can be done in $O(\log n)$ time using $\max\{n^2, M(n)\}$ processors. Hence, path counts can be computed in $O(\log r \log n)$ time using $M(n)$ processors on an EREW PRAM.

4.2. The ϵ-blocking flow algorithm

We present an ϵ-blocking flow algorithm. The algorithm consists of iterations, where each iteration augments the flow. We refer to the difference between the initial capacity and the current flow value of an edge e as the *current capacity* of e. The flow on an edge e does not decrease, and hence, the current capacity of e does not increase. For the ith iteration, we denote by u_e^i $(e \in E)$ the current capacities, by χ_e^i $(e \in E)$ the respective path counts, and by $0 \le f_e^i \le u_e^i$ $(e \in E)$ the flow augmentation computed during the iteration. An iteration amounts to first computing the path counts χ_e $(e \in E)$, with respect to the current capacities, and then computing a valid flow augmentation which is proportional to the path counts and is of maximum possible value. Figure 1 depicts an example for the quantities computed in a single iteration of Algorithm 4.3. The figure shows a capacitated network, the corresponding path counts on the edges, and the resulting flow augmentation.

Algorithm 4.3 [Find an ϵ-blocking flow]

i. $i \leftarrow 0$

ii. Set the initial capacities: For $e \in E$, $u_e^0 \leftarrow \begin{cases} u_e & \text{if } u_e \geq \epsilon \\ 0 & \text{if } u_e \leq \epsilon \end{cases}$

$E \leftarrow E \setminus \{e \in E | u_e^0 = 0\}$

iii. **Repeat:**

(1) Compute the path counts χ_e^i, for $e \in E$, with respect to the capacities u_e^i. Let $\mu_i = \min_{e \in E} u_e^i / \chi_e^i$.

(2) For all $e \in E$, set $f_e^i \leftarrow \mu_i \chi_e^i$, $u_e^{i+1} \leftarrow u_e^i - f_e^i$, if $u_e^{i+1} \leq \epsilon$ set $u_e^{i+1} \leftarrow 0$ ($E \leftarrow E \setminus \{e\}$).

(3) $i \leftarrow i + 1$

Until: t is not reachable from s in E

iv. Output the flow f, where

$$\text{for } e \in E,\ f_e \leftarrow \sum_{j=1}^{i-1} f_e^i \ .$$

4.3. Correctness

For $e \in E$, the quantity $\mu_e^i \equiv u_e^i / \chi_e^i$ is the maximum flow value per path count unit we can put through the edge e without violating the capacity u_e^i. The choice of μ_i as the minimum of these quantities guarantees that for all edges $e \in E$, $f_e^i = \mu_i \chi_e^i \leq u_e^i$.

Proposition 4.2 states that the path counts χ_e obey conservation constraints at the vertices. Hence, the same holds for the quantities $f_e^i = \mu_i \chi_e$ ($e \in E$).

We showed that in each iteration, the values f_e^i ($e \in E$) obey both the conservation constraints at the vertices and the current capacity constraints at all edges, and hence, they constitute a valid flow.

It is easy to see that for $i > 0$, at the end of the ith iteration we have

$$\text{for all } e \in E, \qquad u_e^0 - \epsilon \leq u_e^{i+1} + \sum_{i=0}^{i} f_e^i \leq u_e^0 \ .$$

Additionally, the algorithm terminates at the ith iteration if the removal of all edges ($e \in E$) for which $u_e^0 - \sum_{j=0}^{i} f_e^j \leq \epsilon$ disconnects s and t. The correctness of the algorithm follows:

Proposition 4.4 *When the algorithm terminates, f_e ($e \in E$) computed in step iv is an ϵ-blocking flow.*

Observe that in each iteration, the current capacity of at least one edge is set to 0 (the edge e for which $\mu_i = u_e^i/\chi_e^i$). Hence, the algorithm terminates after at most m iterations. We will prove a much better bound on the number of iterations.

4.4. Bound on the number of iterations

Proposition 4.5 *For every edge $e \in E$, the quantities μ_e^i ($i \geq 0$) are a nondecreasing function of i, and hence, the same is true for $\mu_i = \min_{e \in E} \mu_e^i$ ($i \geq 0$).*

Proof: The current capacities u_e^i are nonincreasing with respect to i. It follows from the definition of the path count χ_e^i of an edge e, that it decreases in at least the same proportion as the current capacity of e (given that all other capacities do not increase). Hence, for every $e \in E$, $u_e^{i+1}/\chi_e^{i+1} \geq u_e^i/\chi_e^i$. ∎

Proposition 4.6

$$\mu_0^{-1} \leq \frac{(n \max_{e \in E} u_e^0)^r}{n\epsilon} .$$

Proof: In a graph of depth r, there are at most n^{r-1} different paths through some edge e. Each path contributes at most $(\max_{e \in E} u_e^0)^r$ to χ_e^i (for all $e \in E$). To bound the denominator, note that for all $e \in E$, $u_e^0 \geq \epsilon$. ∎

Proposition 4.7 *If for some i,*

$$\mu_i^{-1} \leq \epsilon^r / \max_{e \in E} u_e^0 ,$$

then the algorithm terminates at the ith iteration.

Proof: It follows that for all edges $e \in E$,

$$\chi_e^i = u_e^i/\mu_e^i \leq \max_{e \in E} u_e^0/\mu^i \leq \epsilon^r .$$

Hence, $\max_{e \in E} \chi_e^i \leq \epsilon^r$. The latter implies that for every path $p \in \mathcal{P}$, $\prod_{e \in p} u_e^i \leq \epsilon^r$, and hence, there exists an edge $e' \in p$ such that $u_{e'}^i \leq \epsilon$. ∎

Proposition 4.8 *If*

$$j \geq i + \lceil \log(\epsilon^{-1} \max_{e \in E} u_e^i) \rceil ,$$

then $\mu_j/\mu_i \geq 2$.

Proof: Consider the set of edges in a kth iteration

$$E_k = \left\{ e \in E | \mu_k \le \mu_e^k = u_e^k / \chi_e^k \le 2\mu_k \right\} .$$

It follows that for every edge $e \in E_k$, the current capacity decreases by a factor of at least $1/2$ during the kth iteration (that is, for every $e \in E_k$ we have $u_e^{k+1} \le u_e^k/2$). Assume to the contrary that $\mu_j < 2\mu_i$. Consider the edge e such that $\mu_e^j = \mu_j$. Since μ_e^k is nondecreasing with k (see Proposition 4.5), we have $\mu_e^k \le \mu_e^j \le 2\mu_i$ (for $i \le k \le j$). Hence, since the μ_k's are nondecreasing, we have $\mu_e^k \le 2\mu_k$ for $k = i, \ldots, j$. Therefore, $e \in E_k$ for $k = i, \ldots, j$. It follows that

$$u_e^j \le \frac{u_e^i}{2^{(j-i)}} \le \frac{u_e^i}{\epsilon^{-1} \max_{e \in E} u_e^i} \le \epsilon .$$

Hence a contradiction, since in the jth iteration $e \notin E$. ∎

We obtain the following bound on the number of iterations.

Theorem 4.9 *The total number of iterations is at most*

$$1 + \lceil \log(\epsilon^{-1} \max_{e \in E} u_e^0) \rceil \left\lceil (r+1) \log(n\epsilon^{-1} \max_{e \in E} u_e^0) \right\rceil .$$

Proof: Suppose the algorithm terminates at iteration t. It follows from Propositions 4.6 and 4.7 that

$$\mu_{t-1}/\mu_0 \le \frac{\max_{e \in E} u_e^0}{n\epsilon} \left(\frac{n \max_{e \in E} u_e^0}{\epsilon} \right)^r .$$

It follows from Proposition 4.8 that the total number of iterations is at most

$$t \le \lceil \log \epsilon^{-1} \max_{e \in E} u_e^0 \rceil \lceil \log(\mu_{t-1}/\mu_0) \rceil + 1 .$$

The proof follows by combining the two inequalities. ∎

4.5. Complexity

Denote by U_f the width of the fattest s-t path, i.e.,

$$U_f = \max_{p \in \mathcal{P}} \min_{e \in p} u_e .$$

The quantity U_f is a lower bound on the maximum flow and mU_f is an upper bound. The parallel complexity of computing U_f is dominated by the complexity of computing path counts.

We may assume that $\max_{e \in E} u_e \le mU_f$ and $\min_{e \in E} u_e \ge \epsilon$, since an ϵ-blocking flow when setting all capacities larger than mU_f to mU_f, and capacities smaller than ϵ to 0, is also ϵ-blocking with respect to the original capacities.

Proposition 4.10 *Using Algorithm 4.3, an ϵ-blocking flow can be computed in* *i.* $O(r^2 \log m \log^2(\epsilon^{-1} m U_f))$ *time using* $O(m/r)$ *processors, or*

ii. $O(r \log n \log r \log^2(\epsilon^{-1} m U_f))$ *time using* $M(n)$ *processors.*

Proof: From Theorem 4.9, the number of iterations is bounded by

$$1 + \lceil \log(\epsilon^{-1} m U_f) \rceil \left\lceil (r+1) \log(nm\epsilon^{-1} U_f) \right\rceil = O(r \log^2(\epsilon^{-1} m U_f)) \ .$$

The complexity of each iteration is dominated by the computation of path counts. Hence, we obtain the stated time bounds by combining the above with the complexity of computing the path counts (see Subsection 4.1). ∎

Particularly when $\epsilon = o(U_f/m^k)$, we can obtain a faster running time for finding an ϵ-blocking flow by using the following procedure:

Algorithm 4.11 [ϵ-blocking flow when $\epsilon = o(U_f/m)$]
While $U_f > \epsilon$ **Do:**

i. Compute a $\max\{U_f/m, \epsilon\}$-blocking flow (using Algorithm 4.3.)

ii. Update the capacities (subtract the flow values computed in the previous step).

It is easy to verify that when using Algorithm 4.11 we obtain the following:

Theorem 4.12 *An ϵ-blocking flow can be computed in*

i. $O(r^2 \log^2 m \log(\epsilon^{-1} m U_f))$ *time using* $O(m/r)$ *processors, or*

ii. $O(r \log r \log^3 m \log(\epsilon^{-1} m U_f))$ *time using* $M(n)$ *processors.*

We are now ready to present the proof of Theorem 1.1:
Proof of Theorem 1.1: To obtain these bounds we run the modified Dinic algorithm until the depth of the layered networks exceeds $\ell = 4r\epsilon^{-1}$. In each phase we compute an $\hat{\epsilon}$-blocking flow, where $\hat{\epsilon} = \epsilon U_f^0/2m$ (U_f^0 is the width of the fattest path in G). Denote by M the value of the flow when the modified Dinic algorithm terminates and by M^* the value of the maximum flow. We first prove that M and M^* have the desired relation. It follows from the choice of $\hat{\epsilon}$ and ℓ that

$$m\hat{\epsilon} \leq \epsilon M^*/2$$

(since $U_f^0 \leq M^*$), and that

$$(\ell - r)/(\ell + r) \geq (1 - \epsilon/2) \ .$$

By plugging the above in Corollary 3.7, we deduce that

$$M + \epsilon M^*/2 \geq (1 - \epsilon/2) M^* \ .$$

Hence, $M \geq (1-\epsilon)M^*$. We show that the computation can be performed within the desired resource bounds. It is easy to see that all layered networks obtained in the phases of the algorithm have depth at most ℓ, have at most as many nodes and edges as G, and are such that the corresponding quantity U_f is never larger than U_f^0. Hence, it follows from Theorem 4.12 that each phase can be implemented in:

i. $O(\ell^2 \log^2 m \log(\hat{\epsilon}^{-1} m U_f^0))$ time using $O(m/\ell)$ processors, or

ii. $O(\ell \log \ell \log^3 m \log(\hat{\epsilon}^{-1} m U_f^0))$ time using $M(n)$ processors.

To conclude note that there are at most ℓ phases. ∎

4.6. Remarks

Remark 4.13 [blocking only short paths] Consider a network G as above. By modifying Algorithm 4.3 we can find a depth-d blocking flow, i.e., a flow which blocks all paths of length at most d for some $d \leq r$. This is done by using a modified definition of path counts which considers only paths of length at most d. The resource bounds of the modified algorithm are similar, where d replaces r.

Remark 4.14 [Finding an integral flow] The max-flow and blocking flow algorithms presented in this section compute a fractional flow, even when capacities are integral. In Section 5.3 we show how to efficiently round any fractional flow to an integral one in $\mathcal{NC}$ using $O(m)$ processors, and how to obtain an integral $\lfloor\epsilon\rfloor$-blocking flow within the resource bounds given for our ϵ-blocking flow algorithm.

5. Rounding fractional flows

We present an $\mathcal{NC}$ algorithm that considers a network G with integral capacities u_e ($e \in E$) and a fractional s-t flow f_e ($e \in E$) with value M. The algorithm produces an integral flow $\hat{f}_e$ ($e \in E$) of value $\hat{M} \in \{\lceil M - 1/\operatorname{poly}(n)\rceil, \lceil M \rceil\}$ such that $\hat{f}_e \in \{\lfloor f_e \rfloor, \lceil f_e \rceil\}$ for ($e \in E$). We also discuss (i) combining the techniques of this section with the ϵ-blocking flow algorithm of Section 4 to produce an integral $\lfloor\epsilon\rfloor$-blocking flow and (ii) rounding a fractional circulation with at most a small increase in the cost, in networks where costs are associated with the edges.

It should be noted that this section generalizes an $\mathcal{NC}$ algorithm of Goldberg, Plotkin, Shmoys, and Tardos [11], for obtaining an integral minimum cost flow given a fractional near-minimum cost flow on bipartite matching networks (zero-one networks of depth $r = 3$). Their algorithm uses $O(m)$ processors and is based on Gabow's coloring algorithm [8].

5.1. Main iteration

We start with a subroutine for halving the smallest common denominator of the flow values, assuming it is a power of 2. Suppose $\Delta = 2^{-k}$ for some integer k, and the flow f is such that for all $e \in E$, f_e/Δ is integral. The following procedure computes a flow $\hat{f}$, of value at least as large as the value of f, such that for all $e \in E$, $\hat{f}_e/(2\Delta)$ is integral and for all $e \in E$, $|\hat{f}_e - f_e| \leq \Delta$.

Algorithm 5.1 [Double unit size]

i. Mark all edges $e \in E$ for which f_e/Δ is odd.
Denote this set of edges by E'.

ii. E' is such that every vertex except for s and t has an even degree (immediate from conservation constraints).
Find an Eulerian partition of E', ignoring the directions of the edges. We obtained a collection of cycles and an s-t path. (An Eulerian partition is a collection of edge-disjoint cycles and paths such that all edges incident at even degree vertices belong to cycles and every odd degree vertex is the endpoint of exactly one path.)

iii. Partition the set E' into two sets E^+ (edges labeled $+$) and E^- (edges labeled $-$) as follows:

- If $e \in E'$ belongs to an s-t path in the Eulerian partition do as follows. If e is a forward edge in an s-t traversal of the path, assign $e \in E^+$, else, if e is a backward edge, assign $e \in E^-$.
- Edges that belong to cycles are labeled such that for one of the two traversals, all forward edges are in E^+ and all backward edges are in E^-.

iv. Compute a flow $\hat{f}_e$ ($e \in E$) as follows:
If $e \in E^+$, $\hat{f}_e \leftarrow f_e + \Delta$.
If $e \in E^-$, $\hat{f}_e \leftarrow f_e - \Delta$.
If $e \in E \setminus E'$, $\hat{f}_e \leftarrow f_e$.

The computation is dominated by the construction of the Eulerian partition ($O(\log n)$ time using $O(m)$ processors), and the list ranking operations on segments of total length $|E'| = O(m)$ ($O(\log m)$ time using $O(m)$ processors) (see e.g., [18, 19]). It follows that the subroutine can be implemented on an EREW PRAM to run in $O(\log n)$ time using $O(m)$ processors.

5.2. Rounding Algorithm

The following algorithm rounds a flow with general flow values.

Algorithm 5.2 [Obtain integral flow]

i. Round up the flow to units of $\Delta_0 = 2^{-5\lceil \log m \rceil}$.
Note that this may create excesses and deficits at the vertices. The sum of the absolute values of the excesses and deficits is at most $2m\Delta_0 \leq 2m^{-4} \leq 1/4$. The excesses or deficits at s and t are determined as if there was an ingoing edge incident at s and an outgoing edge incident at t with flow value of the s-t flow rounded up to units of Δ_0.

ii. For $k = 0, \ldots, 5\lceil \log m \rceil$ and $\Delta = 2^k \Delta_0$:
If there are excesses and deficits present, run the following modification of Algorithm 5.1. Otherwise, run the unmodified Algorithm 5.1.

Since there are excesses and deficits, the Eulerian partition computed in Step ii of Algorithm 5.1 may contain paths with end points at vertices other than s and t. The labels of edges along such paths are determined in the same way as cycles, where with respect to some traversal, all backward edges are labeled the same and all forward edges are labeled the same. Use the following rules to determine which of the two possibilities to choose from:

- If a path p is such that an end edge is directed into a vertex with a deficit, put the edge in E^+ (if this is true for both end edges of p, arbitrarily choose one).
- Otherwise, if p is such that an end edge is directed out of a vertex with an excess, put the edge in E^+ (if this is true for both end edges of p, arbitrarily choose one).
- Otherwise, put one of the end edges of p in E^-.

Note that (i) at any point, the sum of the excesses and deficits is zero, (ii) the rules to choose the labels guarantee that the sum of the absolute values of excesses and deficits never increases, and (iii) at each vertex, the excess or deficit is an integral number of units of the current value of Δ. Hence, when $\Delta > m^{-4}$, there are no excesses or deficits. Observe that the s-t flow may decrease by at most $2m^{-4}$ during the algorithm, since it may decreases only in iterations where there are excesses or deficits present. It follows that $\hat{M} \geq \lceil M - 2m^{-4} \rceil$.

Proposition 5.3 *The rounding takes $O(\log^2 m)$ time using $O(m)$ processors, on an EREW PRAM.*

Proof: Step i can be performed in $O(\log m)$ time using $O(m)$ processors. Step ii consists of $O(\log m)$ iterations, where each iteration can be performed in $O(\log m)$ time using $O(m)$ processors. ∎

5.3. Computing an integral blocking flow

The rounding algorithm is such that the amount of flow on edges with integral flow is not changed. Hence, when the rounding algorithm is applied to a fractional blocking flow, the result is an integral blocking flow. If the initial fractional flow is ϵ-blocking, the resulting integral flow is $\lceil\epsilon\rceil$-blocking. In this subsection we show how to obtain an $\lfloor\epsilon\rfloor$-blocking flow.

Algorithm 4.3 (ϵ-blocking flow algorithm) produces a fractional ϵ-blocking flow. To obtain an integral $\lfloor\epsilon\rfloor$-blocking flow we use a modified Algorithm 4.3, where at each iteration k, the flow f^k is rounded to an integral flow $\hat{f}^k$ by a rounding algorithm that is presented below. The integral flow $\hat{f}^k$ is used instead of f^k to augment the current flow. Note that the capacities remain integral throughout the execution of the modified Algorithm 4.3.

At each iteration k, the rounding is such that for all $e \in E$, $\hat{f}^k_e \in \{\lceil f^k_e\rceil, \lfloor f^k_e\rfloor\}$. The rounding is done with respect to a parameter η that is determined as follows. Initially, $\eta = \mu_0$. At the beginning of any iteration k, if $2\eta \leq \mu_k$ then we set $\eta = \mu_k$. The rounding will have the following additional property: At least a constant fraction (1/2) of the edges $e \in E$ for which $\mu^k_e \leq 2\eta$ and $u^k_e = 1$ are such that $\hat{f}^i_e = \lceil f^i_e\rceil = 1$. We note that for these edges we have $f^i_e \geq 1/2$. We postpone the discussion of how to achieve this rounding.

We sketch why the bounds of Algorithm 4.3 remain the same. Consider the analysis in Subsection 4.4. It is easy to see that Propositions 4.5, 4.6, and 4.7 hold. We prove the correctness of the following modified version of Proposition 4.8. It is easy to see that the remaining part of the complexity analysis can be carried on with the modified Proposition 4.8 and it yields the same asymptotic bounds.

Proposition 5.4 *(modified Proposition 4.8)*

i. When $\epsilon \geq 1$, for all iterations i, j, if $j \geq i + \lceil\log_{3/2}(\epsilon^{-1}\max_{e\in E} u^i_e)\rceil$, then $\mu_j \geq 2\mu_i$.

ii. When $\epsilon < 1$, if iterations i, j are such that $j \geq i + \lceil\log_{3/2}(\max_{e\in E} u^i_e)\rceil + \lceil\log m\rceil + 1$, η must be updated at least once between the ith and jth iterations.

Proof: We first consider the case where $\epsilon \geq 1$. Since the algorithm eliminates edges of capacity ϵ or less, all capacities are at least 2. Consider the kth iteration ($i \leq k \leq j$) and an edge $e \in E_k$ (where E_k is defined as in the proof of Proposition 4.8). We have $f^k_e \geq u^k_e/2$. Hence, $\hat{f}^k_e \geq \lfloor f^k_e\rfloor \geq u^k_e/3$. Therefore, $u^{k+1}_e \leq \lfloor 2u^k_e/3\rfloor$. The proof follows. We consider the case where $\epsilon < 1$. Assume to the contrary that for some i, j as above, the value of η is not modified. By definition, $\eta \leq \mu_i \leq \cdots \leq \mu_j < 2\eta$. At the kth iteration ($i \leq k \leq j$), consider the set of edges $E_k = \left\{e \in E | \mu^k_e \leq 2\eta\right\}$. Since μ^k_e are nondecreasing with k, we have $E_i \supseteq \cdots \supseteq E_j$. Employing the argument used above and the fact that $\eta \leq \mu_k$, we obtain that for all $e \in E_k$, if $u^k_e \geq 2$ then $u^{k+1}_e \leq \lfloor 2u^k_e/3\rfloor$. Therefore, for $k \geq k' = i + \lceil\log_{3/2}(\max_{e\in E} u^i_e)\rceil$, for all $e \in E_k$, $u^k_e = 1$. It follows from the additional property of the rounding that for $k \geq k'$, $|E_{k+1}| \leq |E_k|/2$. Therefore, $E_j = \emptyset$. This implies that at the jth iteration, for all edges $e \in E$, $\mu^j_e > 2\eta$. This is a contradiction since $\mu_j < 2\eta$ and by definition, there must exist an edge e such that $\mu^j_e = \mu_j$. ∎

What remains is to show how to round the flow such that the resulting integral flow $\hat{f}^k$ has the additional property. It suffices to examine a fractional flow f where edges such that $f_e \geq 1/2$ and $u_e = 1$ are marked. We use a variant of Algorithm 5.2 to round the flow such that at least half of the marked edges are rounded up. The variant runs Algorithm 5.2 as stated, except for the last iteration where $\Delta = 1/2$. The choice of Δ_0 ensures us that the sum of absolute values of excesses and deficit is less than 1/4. Hence, at the iteration where $\Delta = 1/2$ there are no excesses or deficits. Therefore, the Eulerian partition consists of a collection of cycles and possibly, an s-t path. It is easy to see that at the iteration where $\Delta = 1/2$, the flow on each marked edge e is either $\lceil f_e \rceil$ or $\lceil f_e \rceil - 1/2$. Observe that Step ii of Algorithm 5.1 allows us, for each cycle or path, two possible choices of labels of edges that do not violate the conservation constraints. We remark that for the s-t path, one of the choices of labels decreases the s-t flow, but for the purpose of rounding the blocking flow, we allow the net flow to decrease. For each cycle and for the s-t path, we choose the labels that maximize the number of marked edges which are selected to be in E^+.

It follows that using the integral version of the ϵ-blocking flow algorithm, we can compute an exact integral blocking flow when the capacities are integral and polynomial in m, in the following EREW PRAM resource bounds:

i. $O(r^2 \log^3 m)$ time using $O(m/r)$ processors, or

ii. $O(r \log r \log^4 m)$ time using $M(n)$ processors.

5.4. Rounding a flow in a network with costs

Suppose that we are given a fractional circulation f_e $(e \in E)$, possibly with integral excesses and deficits at the vertices (supplies and demands), on a network with integral capacities and real costs c_e $(e \in E)$ associated with the edges. We modify Algorithm 5.2 so that it produces an integral flow $\hat{f}_e$ with the properties $\hat{f}_e \in \{\lfloor f_e \rfloor, \lceil f_e \rceil\}$ $(e \in E)$ and

$$\sum_{e \in E} c_e \hat{f}_e \leq 2mn\Delta_0 \max_{e \in E} |c_e| + \sum_{e \in E} c_e f_e \ ,$$

where Δ_0 is some inverse power of 2 (typically, $\log \Delta_0^{-1} \geq 5\lceil \log m \rceil$). The modified algorithm runs in time $O(\log m \log \Delta_0^{-1})$ and uses $O(m)$ processors. The algorithm uses the input value of Δ_0. Only the excesses and deficits introduced by the rounding are being eliminated. (In other words, the excess/deficit at a vertex is the excess/deficit reduced by the initial integral demand/supply value). In each iteration, we do as follows. For each cycle or path c, there are two ways to label edges with E^+ or E^-. The choice such that $\sum_{e \in p \cap E^+} c_e \leq \sum_{e \in p \cap E^-} c_e$ guarantees that the total cost does not increase as a result of changing the flow on these edges. For cycles, we always choose the labels such that the cost does not increase. Because of the initial rounding, there may be paths with end points at vertices with excesses or deficits. For such paths, if both choices of labels preserve the sum of absolute values of deficits and excesses, we choose the labeling that does not increase the cost. Otherwise, we choose the labeling that decreases the sum of absolute values of deficits and excesses. Note, however, that the cost may increase as a result of the change. Initially, the sum of absolute values of excesses and deficits is at most $2m\Delta_0$. Each unit of excess or deficit may be routed once on a cost-increasing path. The size of each such path is at most n (we may assume that the Eulerian partitions consist of simple paths and cycles although this assumption is not needed in previous subsections). Hence, the total increase in cost resulting from excess/deficit decreasing paths is bounded by $2mn\Delta_0 \max_{e\in E}|c_e|$.

When

$$\Delta_0 = 2^{-5\lceil \log m \rceil} ,$$

the running time is $O(\log^2 m)$ and the resulting integral flow is such that

$$\sum_{e\in E} c_e \hat{f}_e \le 2m^{-4} n \max_{e\in E}|c_e| + \sum_{e\in E} c_e f_e .$$

6. Zero-one and unit networks

Zero-one networks are such that all capacities are in $\{0,1\}$. A unit network is a zero-one network where in addition, each vertex has either a single incident outgoing edge or a single incident ingoing edge. This section is concerned with maximum flow computation on such networks. Even and Tarjan [5] showed that on zero-one (resp., unit) networks, Dinic's algorithm terminates within $O(\min\{n^{2/3}, \sqrt{m}\})$ (resp., $O(\sqrt{n})$) phases. The reason being that if the residual network is such that all augmenting paths are longer than ℓ, the value of the maximum flow is bounded by m/ℓ (resp., n/ℓ). Also note that for any integral flow, the residual networks are of the same form (zero-one or unit, resp.). Hence, after $O(\sqrt{m})$ (resp., $O(\sqrt{n})$) phases, the value of the maximum flow in the residual network is at most $O(\sqrt{m})$ (resp., $O(\sqrt{n})$).

For such networks we can guarantee that if all augmenting paths are long, the value of the maximum flow is small. We obtain parallel maximum flow algorithms as follows. We apply Dinic's algorithm for some number of phases, using the integral version of the algorithm of Section 4 to compute blocking flows. We stop Dinic's algorithm when the depth of the layered network becomes large. To obtain a maximum flow, we apply a parallel version of the Ford-Fulkerson maximum flow algorithm to the residual network. The Ford-Fulkerson algorithm repeatedly finds and saturates augmenting paths until no augmenting path exists. Since the value of the remaining flow is small, the Ford-Fulkerson algorithm does not perform many iterations. A similar balancing between parallel versions of the Ford-Fulkerson algorithm and the Goldberg-Tarjan push-relabel algorithm [13] was previously used by Goldberg, Plotkin, and Vaidya [12] to obtain sublinear time algorithms for such networks.

For the first part (Dinic's algorithm) of the algorithm we use the ϵ-blocking flow algorithm of Section 4. In Subsection 5.3 we showed that when the capacities are integral of size polynomial in m (as is the case for zero-one networks), the algorithm can be modified to find an exact integral blocking flow. For the second part (Ford-Fulkerson's algorithm), we use the fact that a single augmenting path, if one exists, can be found in $O(\log^3 n)$ time with $M(n)$ processors (using parallel BFS) and hence, a flow of value f^* can be found in

$O(f^* \log^3 n)$ time using $M(n)$ processors. By balancing the two parts of the algorithm we determine the number of iterations used and obtain the following bounds. Note that similar bounds for unit and zero-one networks were previously obtained in [12].

Proposition 6.1 *A maximum flow can be computed on an $M(n)$ processor EREW PRAM in:*

i. $O(n^{2/3} \log^{3+1/3} n)$ *time for unit networks,*

ii. $O(m^{2/3} \log^{3+1/3} n)$ *time for zero-one networks,* ($\tilde{O}((\sum_{e \in E} u_e)^{2/3})$ *for general networks with integral capacities).*

Proof: The running times are obtained by first running Dinic's algorithm, using the blocking flow algorithm of Section 4, until the depth of the layered network exceeds some parameter ℓ. When capacities are small, this first part of the algorithm takes $O(\ell^2 \log \ell \log^3 n)$ parallel time, using $M(n)$ processors (see Proposition 4.10). (Note that we may assume that capacities are integral and at most polynomial in n.) The second part consists of running a parallel version of the Ford-Fulkerson algorithm. The running time is $O(f^* \log^3 n)$, using $M(n)$ processors, where f^* is the value of the remaining flow. For part i we choose $\ell = (n/\log n)^{1/3}$ and for part ii we choose $\ell = (m/\log n)^{1/3}$. ∎

7. Approximate bipartite matching in parallel

A matching in a graph is a set of edges such that no two edges are incident at the same vertex. The maximum cardinality bipartite matching (MCBM) problem is to find, in a bipartite graph, a matching of maximum size. The problem can be stated as an integral maximum flow problem on a unit network of depth 3. The corresponding network is such that the set of saturated edges of any integral flow turns out to be a matching of cardinality which equals the value of that flow. Hence, MCBM can be solved using Dinic's algorithm in $O(m\sqrt{n})$ time [5] (see also [17] for an algorithm stated in terms of alternating paths). A blocking flow computation on a unit network can be done in $O(m)$ time [5], and hence, it follows from Corollary 3.4, that an approximate matching can be found in $O(\epsilon^{-1} m)$ time.

As for the parallel complexity of the MCBM problem, it is still not known whether the problem is in $\mathcal{NC}$. It is known, however, to be in $\mathcal{RNC}$. The first $\mathcal{RNC}^3$ algorithm was obtained by Karp, Upfal, and Wigderson [20]. Their algorithm uses $O(n^{6.5})$ processors. A later $\mathcal{RNC}^2$ algorithm by Mulmuley, Vazirani, and Vazirani [23] uses $O(n^{3.5} m)$ processors. The best known $\mathcal{RNC}$ processor bound of $O(nM(n))$ (in $\mathcal{RNC}^3$) was achieved by Galil and Pan [10]. Pippenger, and Valiant [21] gave an $\mathcal{NC}$ algorithm for the special case of regular bipartite graphs, and Miller and Naor [22] gave an $\mathcal{NC}$ algorithm for planar bipartite graphs. An algorithm by Goldberg, Plotkin, and Vaidya [12] solves the MCBM problem in $O(n^{2/3} \log^3 m)$ time using $M(n)$ processors. An interior point based algorithm by Goldberg, Plotkin, Shmoys, and Tardos [11] solves the problem in $\tilde{O}(\sqrt{m})$ time using $O(m^3)$ processors.

algorithm	*time*	*processors*
deterministic, approximation to $(1-\epsilon)$		
Fisher, Goldberg, Plotkin [6]	$O(\log^3 m)$	$O(n^{2\epsilon^{-1}})$
	$O(\epsilon^{-1} m)$	1
This paper	$O(\epsilon^{-3}\log^2 m \log(m\epsilon^{-1}))$	$O(m)$
	$O(\epsilon^{-2}\log^3 m \log \epsilon^{-1} \log(m\epsilon^{-1}))$	$O(M(n))$
deterministic		
Hopcroft, Karp [17]	$O(m\sqrt{n})$	1
Goldberg, Plotkin, Shmoys, Tardos [11]	$\tilde{O}(\sqrt{m})$	$O(m^3)$
Goldberg, Plotkin, Vaidya [12]	$O(n^{2/3}\log^3 m)$	$O(M(n))$
Gabow, Tarjan [9]	$O(n\log^2 n)$	$O(m/(n^{1/2}\log n))$
Shiloach, Vishkin [25]	$O(n^{3/2}\log n)$	$O(m/n)$
randomized		
Karp, Upfal, Wigderson [20]	$O(\log^3 m)$	$O(n^{6.5})$
Mulmuley, Vazirani, Vazirani [23]	$O(\log^2 m)$	$O(n^{3.5} m)$
Galil, Pan [10]	$O(\log^3 n)$	$O(nM(n))$

Table 1: Parallel maximum cardinality bipartite matching results

In [6], Fisher, Goldberg, and Plotkin gave an $\mathcal{NC}$ algorithm which finds a matching in a graph which is at least $(1-\epsilon)$ of the maximum cardinality matching. They first showed that if a matching is such that there are no augmenting paths of length smaller than ℓ, then the matching is of cardinality at least $(1-1/\ell)$ of maximum. Their algorithm is roughly based on examining all paths of length at most ℓ, and hence uses $n^{O(\ell)}$ processors. As a corollary of Theorem 1.1, we obtain an $\mathcal{NC}$ algorithm which finds a bipartite matching of cardinality $(1-1/\operatorname{polylog} n)$ of optimal and uses only a linear $O(m)$ number of processors. A similar result was obtained very recently by Spencer [26], by using a modification of the parallel exact matching algorithm of Goldberg, Plotkin, and Vaidya [12]. The results mentioned above are summarized in Table 1.

Theorem 7.1 *Bipartite matching of cardinality* $(1-\epsilon)$ *of maximum can be computed in:*

i. $O(\epsilon^{-3}\log^2 m \log(m\epsilon^{-1}))$ *time using* $O(m)$ *processors, or*

ii. $O(\epsilon^{-2}\log^3 m \log \epsilon^{-1} \log(m\epsilon^{-1}))$ *time using* $O(M(n))$ *processors.*

Proof: The algorithm is as follows. First we translate the bipartite matching instance to an s-t maximum flow problem on a unit network of depth $r=3$. The maximum flow on this network equals the maximum cardinality of a matching in the original problem. An integral flow corresponds to a matching. We first find a fractional flow that approximates the maximum flow to within a factor of $(1-\epsilon)$. This can be done within the time bound stated in Theorem 1.1. The finishing step amounts to rounding the flow to an integral flow of at least the same value. This was previously done by Goldberg, Plotkin, Shmoys, and Tardos in [11], who gave an $O(\log^2 n)$ time algorithm which uses $O(m)$ processors. ∎

8. Open problems

To conclude, we discuss some issues and open problems that arise from this work.

Achieving better bounds. We presented an $O(M(n))$ processors $\tilde{O}(r)$ time algorithm for approximate blocking flow (exact for polynomial capacities). It is still open whether we can improve on this, that is, find an (approximate) blocking flow in $O(r^{1-\epsilon})$ deterministic parallel time for some $\epsilon > 0$. Such an algorithm would have interesting implications, one of which is that a flow with no augmenting paths shorter than r could be found in $O(r^{2-\epsilon})$ deterministic parallel time. (We remark that computing such a flow within the latter bounds might be an easier task than the first one.) The latter bound, even for the special case of unit networks, would yield a faster time bound than currently known for computing maximum cardinality bipartite matching deterministically, in parallel, using polynomial work.

Extension to minimum cost flow. We ask whether our results can be extended to finding near-minimum cost flow more efficiently on small depth networks, or more specifically, to minimum weight bipartite matching. One problem is that the augmenting path argument in the proof of Proposition 3.1 does not seem to carry over for minimum cost flows.

Weighted path counts. In this paper we introduced the notion of path counts and used path counts weighted by capacities to compute a flow augmentation in each iteration of the blocking flow algorithm. We ask whether we can benefit by considering path counts weighted by other "costs" associated with the edges, which could possibly be dependent on the capacities and other parameters. Weighted path counts keep the properties of being efficiently computable in parallel and being conserved at vertices. Consider, for simplicity, a zero-one network of depth r. Let $\boldsymbol{c} = \{c_e | e \in E\}$ be nonnegative weights on the edges, and denote by χ_e $(e \in E)$ the weighted path counts (relatively to the weights c_e). Each such system of weights, corresponds to a flow $f(\boldsymbol{c})$ where

$$f_e(\boldsymbol{c}) = \frac{\chi_e}{\max_{e' \in E} \chi_{e'}} \quad (e \in E) .$$

Denote the value of $f(\boldsymbol{c})$ by $M(\boldsymbol{c})$. We ask how close to the maximum flow M^* is the value of the maximum flow that has the form $f(\boldsymbol{c})$ for some nonnegative set of costs $\boldsymbol{c}$. For unit networks we have $M^* = \max_{\boldsymbol{c}} M(\boldsymbol{c})$. We suggest an approach for an algorithm that directly approximates the maximum flow without breaking the computation to blocking flow phases. The approach is to start with some initial costs and to keep updating them until $M(\boldsymbol{c})$ is close to M^* for the current weights $\boldsymbol{c}$.

Acknowledgements. The author would like to thank Andrew Goldberg, David Johnson, Seffi Naor, Tomek Radzik, Peter Shor, Vijay Vazirani, and Alex Wang for relevant discussions and pointers to the literature, and the anonymous referee for careful reading and many helpful comments.

References.

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

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

[3] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9:251–280, 1990.

[4] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[5] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4:507–518, 1975.

[6] T. Fisher, A. V. Goldberg, and S. A. Plotkin. Approximating matching in parallel. Manuscript, 1991.

[7] L. R. Ford Jr. and D. R. Fulkerson. *Flows in networks*. Princeton Univ. Press, Princeton, NJ, 1962.

[8] H. N. Gabow. Using Euler partitions to edge-color bipartite multi-graphs. *Int. J. Comput. Inform. Sci.*, 5:345–355, 1976.

[9] H. N. Gabow and R. E. Tarjan. Almost-optimum speed-ups of algorithms for bipartite matching and related problems. In *Proc. 20th Annual ACM Symposium on Theory of Computing*, pages 514–527. ACM, 1988.

[10] Z. Galil and V. Pan. Improved processor bounds for combinatorial problems in RNC. *Combinatorica*, 8:189–200, 1988.

[11] A. V. Goldberg, S. A. Plotkin, D. B. Shmoys, and É. Tardos. Interior point methods in parallel computation. In *Proc. 30th IEEE Annual Symposium on Foundations of Computer Science*, pages 350–355. IEEE, 1989.

[12] A. V. Goldberg, S. A. Plotkin, and P. M. Vaidya. Sublinear time parallel algorithms for matching and related problems. In *Proc. 29th IEEE Annual Symposium on Foundations of Computer Science*, pages 174–185. IEEE, 1988.

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

[14] A. V. Goldberg and R.E. Tarjan. A parallel algorithm for finding a blocking flow in an acyclic network. *Information Processing Let.*, 31:265–271, 1989.

[15] A. V. Goldberg and R.E. Tarjan. Finding minimum cost circulations by successive approximation. *Math. of Oper. Res.*, 15:430–466, 1990.

[16] L. M. Goldschlager, R. A. Shaw, and J. Staples. The maximum flow problem is logspace complete for P. *Theor. Comp. Sci.*, 21:105–111, 1982.

[17] J. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Comput.*, 2:225–231, 1973.

[18] J. JáJá. *Parallel algorithms.* Addison-Wesley Publishing Co., Reading, MA, 1992.

[19] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwenm, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 17, pages 869–941. MIT Press/Elsevier, Cambridge, MA, 1991.

[20] R. M. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random NC. *Combinatorica*, 6:35–48, 1986.

[21] G. F. Lev, N. Pippenger, and L. G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Trans. Computers*, C-30:93–100, 1981.

[22] G. L. Miller and J. Naor. Flow in planar graphs with multiple sources and sinks. In *Proc. 30th IEEE Annual Symposium on Foundations of Computer Science*, pages 112–117. IEEE, 1989.

[23] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

[24] V. Pan and J. Reif. Efficient parallel solution of linear systems. In *Proc. 17th Annual ACM Symposium on Theory of Computing*, pages 143–152. ACM, 1985.

[25] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. *J. Alg.*, 3:128–146, 1982.

[26] T. H. Spencer. Parallel approximate matching. Technical Report UNO-CS-TR-92-5, University of Nebraska at Omaha, 1992.

[27] R. E. Tarjan. *Data structures and network algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.