UNIVERSAL BUFFERLESS PACKET SWITCHING*

COSTAS BUSCH†, MALIK MAGDON-ISMAIL‡, AND MARIOS MAVRONICOLAS§

Abstract. A packet-switching algorithm specifies the actions of the nodes in order to deliver packets in the network. A packet-switching algorithm is *universal* if it applies to any network topology and for any batch communication problem on the network. A long standing open problem has concerned the existence of a universal packet-switching algorithm with near optimal performance guarantees for the class of *bufferless* networks where the buffer size for packets in transit is zero. We give a positive answer to this question. In particular, we give a universal bufferless algorithm which is within a poly-logarithmic factor from optimal for arbitrary batch problems:

$$\mathcal{T} = O\left(\mathcal{T}^* \cdot \log^3(n+N)\right),$$

where $\mathcal{T}$ is the packet delivery time of our algorithm, $\mathcal{T}^*$ is the optimal delivery time, n is the size of the network, and N is the number of packets.

At the heart of our result is a new *deterministic* technique for constructing a universal bufferless algorithm by emulating a store-and-forward algorithm on a transformation of the network. The main idea is to replace packet buffering in the transformed network with packet circulation in regions of the original network. The cost of the emulation on the packet delivery time is proportional to the buffer sizes used by the store-and-forward algorithm. We obtain the advertised result by using a store-and-forward algorithm with logarithmic sized buffers. The resulting bufferless algorithm is constructive and it can be implemented in a distributed way.

Key words. optimal scheduling, routing, graph decomposition, deterministic bufferless emulation

AMS subject classifications. 68W15, 68W20, 90B20, 90B35, 90B36

1. Introduction.

1.1. Motivation.

In a communication network, two or more packets collide when they wish to follow the same link at the same time. Typically, some of the colliding packets are stored in a buffer at the node where the collision occurs, until the collision is resolved (*store-and-forward* networks). Here, we examine the case where such buffers are unavailable (*bufferless* networks). At the same time, we do not allow packets to be dropped in collisions. Since buffers are unavailable and packets cannot be dropped, colliding packets must be deflected to neighboring nodes. This behavior of packets in a collision has led to communication algorithms on bufferless networks becoming known as *hot-potato* or *deflection* algorithms; here, we will simply call them *bufferless*. Bufferless algorithms are of practical interest since in optical networks the packets are propagated as light-waves which are hard to buffer [45].

A packet switching algorithm specifies the actions that nodes in the network follow to deliver the packets. A packet switching algorithm is *universal* if it applies to any network topology and can solve any batch problem on it, where an arbitrary set of packets has to be delivered in the network. Universal store-and-forward algorithms

*A preliminary version of this paper appears as “Universal Bufferless Routing”, in Proceedings of the 2nd Workshop on Approximation and On-line Algorithms (WAOA) (in conjunction with ALGO 2004), LNCS 3351, pp 239–252, Bergen, Norway, September 2004.

† Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180, USA; buschc@cs.rpi.edu

‡ Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180, USA; magdon@cs.rpi.edu

§ Department of Computer Science, University of Cyprus, P. O. Box 20537, Nicosia CY-1678, Cyprus; mavronic@ucy.ac.cy. Partially supported by the EU within the 6th Framework Programme under contract 001907 “Dynamically Evolving, Large Scale Information Systems” (DELIS).

with optimal performance guarantees exist [24, 35, 33, 40, 43]. A long standing and important open problem is to determine whether there exists a universal bufferless algorithm with performance close to that of store-and-forward algorithms. Here, we solve this problem in the affirmative by giving the *first* known universal bufferless algorithm with near-optimal performance. We analyze the performance of our algorithm for batch problems on a synchronous network model, which we describe now.

1.2. Network Model. The communication network is a connected, unweighted and undirected graph $G = (V, E)$, where $|V| = n$. In a *synchronous* network, time is divided into a sequence of discrete time steps. Edges are bidirectional and may be traversed by at most two packets at a time step, one packet in each direction.

At every time step, a node processes the incoming packets, and then sends them to adjacent nodes. In store-and-forward networks, each node has three kinds of buffers: (i) an *injection buffer*, which stores the packets to be injected into the node (when the node is a packet source), (ii) *incoming edge-buffers*, of size one for every incident edge, which will store any packet received along the respective edge, (iii) *outgoing edge-buffers*, for every incident edge, which is the actual buffer for packets in transit. At every time step the node takes the packets from the incoming and injection buffers and either forwards them along incident links or places them in the outgoing buffers. If the outgoing buffer is full, then packets are dropped.

In bufferless networks, there are no outgoing edge-buffers; in other words, all outgoing edge-buffers have size zero. Further, packets may not be dropped. Thus, after the packet is injected into the network, it must traverse some edge at every time step (until it is absorbed). Preferably, the traversed edge brings the packet closer to the destination. However, due to collisions, this is not always possible and the packet may be sent on an alternate edge taking it further from the destination; this event is called *deflection*.

In our model, bufferless networks still have the injection and incoming buffers. The incoming buffers help to process the incoming packets. The injection buffer is needed when a node has to inject packets and there are no available edges. So, the distinction between store-and-forward and bufferless networks is in the outgoing buffers which hold packets in transit.

1.3. Batch Problems. We measure the efficiency of our bufferless algorithm on *batch problems* [32]. In a batch problem, we are given an arbitrary set of packets with the objective to deliver them to their destinations. Let $Q = (G, \Pi, \mathcal{S})$ denote a batch problem on graph G, for a set of N packets $\Pi = \{\pi_1, \pi_2, \ldots, \pi_N\}$. Each packet π_i has source s_i and destination t_i. The set $\mathcal{S}$ contains all the pairs of sources and destinations for the respective packets; so $\mathcal{S} = \{(s_1, t_1), (s_2, t_2), \ldots, (s_N, t_N)\}$.

We say that a set of paths $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$ *satisfies* batch problem Q if each path p_i is a path in G from the source s_i to the destination t_i of packet π_i. Typically, a packet switching algorithm solves a batch problem Q by first selecting a set of paths $\mathcal{P}$ that satisfy Q (*routing*), and then sending the packets along the paths (*scheduling*). The *delivery time* of the packet switching algorithm is the number of time steps that elapse between the first packet injection and the last packet absorbtion at its destination.

It is useful to define some properties associated with a set of paths $\mathcal{P}$. A path $p \in \mathcal{P}$ could be specified either as a sequence of nodes, or as a sequence of edges, and the length of the path, $|p|$, is the number of edges in the path. The *edge congestion* C is the maximum number of paths in $\mathcal{P}$ that use an edge in G; similarly, the *node congestion* $\overline{C}$ is the maximum number of paths that use a node in G; the *dilation* D

is the maximum path length, $\max_{p_i \in \mathcal{P}} |p_i|$. Since at most one packet may traverse an edge in a particular direction during any time step, a lower bound on the delivery time is given by $\Omega(C + D)$. Any packet switching algorithm, either bufferless or store-and-forward, obeys this lower bound.

Given a set of paths with edge congestion C and dilation D, there exist store-and-forward scheduling algorithms that deliver the packets in time $O(C + D)$ (plus logarithmic terms), which are optimal within constant factors for the given paths [24, 35, 33, 40, 43]. Let $\mathcal{P}^*$ denote the optimal set of paths which minimize $C + D$ for a given batch problem Q. Using the optimal paths, the packets can be delivered in optimal time (within constant factors) in store-and-forward networks. A long standing open question is whether one can achieve near optimal delivery time in bufferless networks as well. We answer this question in the affirmative.

1.4. Contribution. We show that for any batch problem Q on an arbitrary bufferless network G, a delivery time within logarithmic factors from optimal can be achieved. In particular, we give a randomized algorithm which does so, given the optimal paths:

THEOREM 1.1. *With probability* $1 - O((n+N)^{-\lambda})$*, for some constant* $\lambda > 0$*, any batch problem* Q *with* N *packets on an arbitrary bufferless network* G *with* n *nodes can be solved with delivery time* $O(\mathcal{T}^* \cdot \log^3(n+N))$*, where* $\mathcal{T}^*$ *is the optimal delivery time for* Q *(with or without buffers).*

In order to obtain this result, we actually prove that for any set of paths $\mathcal{P}$ with congestion C and dilation D, the packets can be delivered within time $O((C + D) \cdot \log^3(n + N))$. Thus, using the optimal set of paths $\mathcal{P}^*$, we obtain Theorem 1.1.

Our algorithm is universal, since it applies to arbitrary network topologies. It also applies to arbitrary batch problems. Given the set of paths $\mathcal{P}^*$, the algorithm is also constructive and can be implemented in a distributed manner. We do not address the issue of constructing the good (optimal) set of paths $\mathcal{P}^*$, which is an active area of research [5, 6, 44, 49]. Our focus is on the fundamental difference between buffered versus bufferless packet switching, which we show is small. We continue by describing the technique used in our algorithm.

1.5. Approach. Consider a batch problem $Q = (G, \Pi, \mathcal{S})$ in a bufferless network G. Let $\mathcal{P}$ be a set of paths that satisfy Q with edge congestion C and dilation D. Our goal is to deliver the packets in time $\mathcal{T} = O((C + D) \cdot \log^3(n + N))$. This is sufficient for proving Theorem 1.1.

If the packets are to be sent without any collisions, then there are no deflections and the only parameter that needs to be determined is the injection time of the packets. In [22], it is shown that it is an NP-hard problem to approximate efficiently the optimal injection times in a collision-free packet scheduling (by a reduction from the *vertex coloring* problem). Thus, packets need to be deflected in order to obtain a near-optimal solution in polynomial time. However, with deflections, it is hard to preserve packets along specific paths, since a packet may need to deviate from its path in order to give priority to packets that make progress.

Our approach to solve the problem is to control the packet deflections while the packets follow the paths in $\mathcal{P}$. We restrict the deflections in some particular areas of the network which are close to the original paths. By controlling the deflections in those areas, we effectively obtain a new set of paths $\widehat{\mathcal{P}}$ in G along which we can send the packets in a collision-free manner and with delivery time $\mathcal{T}$. We implicitly obtain the new set of paths and the collision-free schedule using a new technique that emulates a store-and-forward algorithm. There are three main steps in the emulation, which we describe next.

1. Creation of store-and-forward network G':. We transform the bufferless network G to a new store-and-forward network G'. Instead of separate outgoing edge-buffers, each node in G' has a unique *outgoing node-buffer* of size γ to store all the packets in transit. We divide the graph G into *regions* which are pairwise edge-disjoint connected components each consisting of about γ edges. Each node in G' corresponds to a region in G. (G' is also called the *region graph* of G, see Figure 1.1.)

The set of paths $\mathcal{P}$ in G is translated to set of paths $\mathcal{P}'$ in G'. If path $p \in \mathcal{P}$ uses an edge e in a region R of G, then in the respective path $p' \in \mathcal{P}'$, edge e is mapped to the node that represents R in G'. The final path p' is obtained by removing any cycles. We observe that the set of paths $\mathcal{P}'$ have node congestion $\overline{C}' = O(\gamma C)$, since at most γC paths of G are mapped to a single node in G'. Further, the dilation is $D' = O(D)$, since a path in G shrinks in G'.

2. Store-and-forward scheduling in G':. We execute a store-and-forward scheduling algorithm for the packets Π in G' using the paths in $\mathcal{P}'$. The scheduling algorithm has delivery time $\mathcal{T}' = O(C + D)$ and uses node-buffers of size $\gamma = O(\log(n + N))$. The algorithm is randomized and efficiently delivers the packets with high probability.

3. Creation of set of paths $\widehat{\mathcal{P}}$ in G:. The store-and-forward schedule in G' is translated back to the original bufferless network G to give the set of paths $\widehat{\mathcal{P}}$ and a collision-free schedule for the packets. The translation is achieved implicitly with a deterministic bufferless emulation of the store-and-forward algorithm in G'. Each time step of the store-and-forward scheduling algorithm in G' is translated to a sequence of $O(\gamma^2 \cdot \log n)$ time steps in G. The main trick is to emulate the buffering. If in a time step a packet is buffered in a node in G', then the same packet in G circulates on an Euler tour of the edges in the corresponding region. Since a buffer in G' may hold multiple packets (but no more than γ), all those packets will circulate one after the other on the edges in the same region; recall that there are at least γ edges in each region. If in a time step a packet moves from one node to another node in G', then the packet moves from the first respective region to the next region in G. If a packet is injected (resp. absorbed) in G, the packet is injected (resp. absorbed) in the source (resp. destination) of the respective region.

During the emulation, the packets may appear to be deflected, while they are in fact circulating on the Euler tours in the region. By concatenating for each packet the respective Euler tours, we implicitly obtain the set of paths $\widehat{\mathcal{P}}$ on which the deflections in effect determine a collision-free schedule.

The cost of the emulation on the delivery time is a factor $O(\gamma^2 \cdot \log n)$ with respect to the store-and-forward schedule; that is, the resulting delivery time is $\mathcal{T} = O(\mathcal{T}' \cdot \gamma^2 \cdot \log n)$. This gives the desired delivery time of $\mathcal{T} = O((C+D) \cdot \log^3(n+N))$, which holds with high probability. We emphasize that the randomization is due to the store-and-forward algorithm, while the emulation is deterministic. Further, if the nodes know the graph G and the parameters C and N, then the resulting bufferless algorithm can be implemented in a distributed way.

1.6. Related Work. There are no previously known results for *universal* bufferless packet switching algorithms with near-optimal delivery time guarantees. However, there are efficient algorithms for specific bufferless models and architectures, which we summarize below.

In *hot-potato* algorithms, packets are deflected to available links in a collision [8]. Our model of bufferless algorithms is based on the hot-potato model, with the

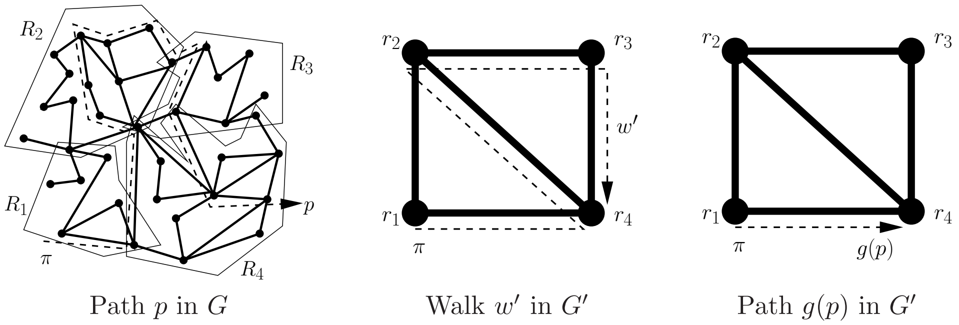


FIG. 1.1. *An example of the decomposition to the region graph.*

significant exception that packets are deflected on particular available edges specified by the emulation, not on any available edge as is typical in hot-potato algorithms. This enables us to control the positions of the packets and implicitly obtain the aforementioned paths $\widehat{\mathcal{P}}$ and a collision-free schedule on them. Hot-potato algorithms have been extensively studied for a variety of architectures such as the mesh and torus [7, 9, 10, 14, 16, 18, 19, 20, 25, 26, 29, 30, 31, 39, 48], hypercubes [13, 15, 26, 28, 42], trees [23, 46], vertex-symmetric networks [36], and leveled networks [12, 17, 21]. Typically, by allowing packets to deviate slightly from their pre-selected paths, one obtains delivery times that are within poly-logarithmic factors of optimal.

In *direct* (*collision-free*) packet scheduling, packets follow their paths without buffering and without any collisions, [4, 50, 22]. Busch *et al.* [22] give a comprehensive study of direct scheduling where they give a universal $O(C \cdot D)$ centralized algorithm, and near-optimal algorithms for the tree, mesh, butterfly and hypercube.

Wormhole algorithms are similar to direct algorithms, although here, packets occupy more than one edge [24, 27]. Cypher *et al.* [24] give a randomized, universal distributed wormhole algorithm with delivery time $O(L \cdot C \cdot D)$, where L is the length of the packet; this bound can be improved if the edges have higher bandwidths. A dual to direct scheduling is *time-constrained* scheduling, where the task is to schedule as many packets as possible within a given time frame [1, 2]. In the related class of *matching* routing algorithms, packets are swapped at adjacent nodes. Under this model, permutation problems on trees have been studied in [3, 41, 51].

There are two variants of store-and-forward algorithms. Those that use outgoing buffers on every edge (*edge-buffers*) and those that use a single outgoing buffer on every node (*node-buffers*). For non-bounded degree networks, these variants may not be equivalent, since one model does not necessarily translate to the other. The existence of universal store-and-forward scheduling algorithms with optimal delivery time $O(C + D)$ (plus additive logarithmic terms) and constant size edge-buffers was first established in the seminal work of Leighton, Maggs and Rao [35]. Scheideler [47] showed that edge-buffers of size 2 are sufficient. These results are non-constructive. Thereafter, the main focus has been on constructive algorithms with optimal delivery time $O(C + D)$ [11, 33, 37, 40, 43]. These algorithms use large buffers (proportional to the congestion C). Leighton *et al.* [35] give a universal distributed algorithm that uses edge-buffers of size $O(\log ND)$ and has delivery time $O(C + D \log ND)$. Cypher *et al.* [24] give an algorithm with edge-buffers of size $O(\log CD)$ and slightly better delivery time. There are no better constructive results known for arbitrary networks that achieve smaller buffer sizes.

Our bufferless algorithm is based on emulating the universal distributed distributed store-and-forward algorithm in Leighton *et al.* [35]. Here, we analyze the delivery time of this store-and-forward algorithm in terms of the node congestion $\overline{C}$ and bound the node-buffer requirements, which is necessary for determining the performance of the bufferless emulation.

1.7. Paper Outline. In Section 2, we give a graph decomposition into regions which is fundamental to our bufferless algorithm. Next, in Section 3, we discuss store-and-forward algorithms, and introduce a simple randomized algorithm using node-buffers. In Section 4, we show how to emulate store-and-forward algorithms in a bufferless manner using the regions obtained by the graph decomposition in Section 2. We apply the emulation on the randomized store-and-forward algorithm to obtain near-optimal universal bufferless algorithm in Section 5. We finish with a discussion and future directions in Section 6.

2. Regions. The bufferless emulation uses regions in the graph to simulate buffering. To construct these regions, we will need to decompose the connected graph G into connected components of approximately a specified size.

2.1. Graph Decomposition. Let $G = (V, E)$ be a connected, unweighted, and undirected graph. Let F be a subset of the edges in E. The subgraph induced by F is the graph $H = (U, F)$, where U is the union of all vertices in V that are incident with edges in F. The edge set F is *connected* if the induced subgraph H is connected.

Definition 2.1. *A connected decomposition of G is a* partition *of the edges in E into a collection of disjoint sets $\{E_1, E_2, \dots, E_k\}$ such that $\cup_{i=1}^{k} E_i = E$ and every E_i is connected.*

We refer to the E_i's as the *connected edge sets* or *regions* in the decomposition, and call the number of edges in E_i, $|E_i|$, the *size* of E_i. Notice that the subgraphs $H_1 = (V_1, E_1), \dots, H_k = (V_k, E_k)$ induced by the edge sets may have overlapping vertex sets. We say that E_i is *connected* to E_j if and only if $V_i \cap V_j \neq \emptyset$. Notice that if E_i is connected to E_j, then $E_i \cup E_j$ is a connected edge set.

An $[\alpha, \beta]$*-partition* of G (if it exists) is a connected decomposition $\{E_1, \dots, E_k\}$ of G, such that $\alpha \leq |E_i| \leq \beta$ for $i = 1, \dots, k$. Notice that if $\alpha \approx \beta$, then an $[\alpha, \beta]$-partition decomposes G into connected edge sets of size approximately equal to α. Our first task is to show that such approximate decompositions exist for any connected graph. Our proof is constructive, hence, it can be converted to an algorithm.

The following lemma will be instrumental in the proof. Essentially, it states that a connected graph can be decomposed into two large connected edge sets.

Lemma 2.2. *Let $k \geq 2$. Any connected graph $G = (V, E)$ with $|E| \geq 3k - 2$ can be decomposed into two disjoint connected edge sets each of size at least k.*

Proof. Using a depth first search, determine a connected edge set F with $|F| = 2k - 2 \geq k$. Note that $|E \setminus F| \geq k$. $E \setminus F$ is composed of a number of connected edge sets (called *satellites*) $\alpha_1, \alpha_2, \alpha_3 \dots$, each of which is not connected to any other, but all of which are connected to F. The situation is illustrated in Figure 2.1. Let α_1 be the largest such satellite edge set of F. If $|\alpha_1| \geq k$ then α_1 and $F \cup \alpha_2 \cup \alpha_3 \cdots$ are both connected edge sets that have size $\geq k$, so we are done. We only need to consider the case where $|\alpha_1| \leq k - 1$. We will show how to replace $|F|$ with another edge set F' of the same size, and whose largest satellite α_1' will have size at least $|\alpha_1| + 1$. We can thus repeat this argument until the size of α_1 is at least k, concluding the proof. We now show how to construct F'.

Suppose that $|\alpha_1| \leq k-1$, in which case there is at least one other satellite α_2. Let u and v be common nodes of α_1 and F, and of α_2 and F respectively. (Note that these nodes must exist, and they are different since F is connected to both these satellites.) Let e be an edge in F incident with u, and let f be an edge in α_2 incident with v (as shown in Figure 2.1). Increase the size of α_1 to $|\alpha_1|+1$ by adding e to it (and removing e from F). Note that α_1 remains connected. If $F \setminus e$ is a connected edge set, then add f to $F \setminus e$ to get F'. Note that $|F'| = |F|$. The edge set $\alpha_1 \cup e$ is now a connected edge set of size $|\alpha_1|+1$ which is part of the largest satellite of F' (note that while adding e to α_1, we may have connected α_1 to some other satellite). Thus the largest satellite α_1' of F' has size at least $|\alpha_1|+1$.

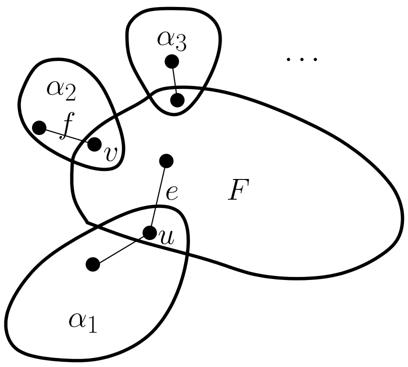


Fig. 2.1. *F and its satellites α_i.*

The only remaining case to consider is that including e into α_1 disconnects F, so e is a bridge in F connecting two connected edge sets $F_1, F_2 \subset F$. The situation is illustrated in Figure 2.2, where we have merged F_1 and F_2 with their respective satellites to get edge sets γ_1 and γ_2 as illustrated. Note that since $|\alpha_1 \cup e| \leq k$, $|\gamma_1| + |\gamma_2| \geq 2k-2$. If neither $|\gamma_1| \geq k$ nor $|\gamma_2| \geq k$, this implies that $|\gamma_1| = |\gamma_2| = k-1$. In this case, merge γ_2 with e to form a connected edge set of size k. The remaining edges form a connected edge set of size at least $2k-2 \geq k$, so we are done. Thus, suppose that one of γ_1 or γ_2 has size $\geq k$; without loss of generality, suppose that it is γ_1. Now consider $\alpha_1' = \gamma_2 \cup \alpha_1 \cup e$. There are two cases: $|\alpha_1'| \geq k$, and we are done; or $|\alpha_1| < |\alpha_1'| < k$, in which case $|\gamma_1| \geq 2k-1$, so that F_1 together with its satellites contains more than $2k-2$ edges. We construct F' from F_1 by adding edges from its satellites (while keeping it connected) until $|F'| = 2k-2$. To conclude, note that the largest satellite of F' must have size at least $|\alpha_1'| \geq |\alpha_1| + 1$. ▯

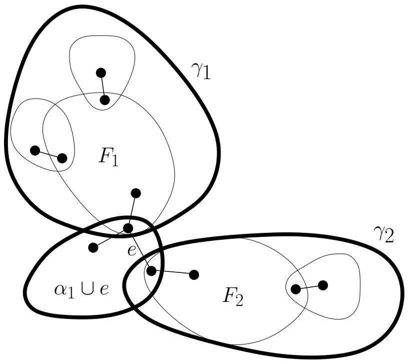


Fig. 2.2. *e is a bridge in F.*

Using an induction argument and Lemma 2.2 we will show that there always exists an $[\alpha, \beta]$-partition with $\beta = \Theta(\alpha)$.

Theorem 2.3 (Existence of a $[k, 3k-3]$-partition). *Let $G = (V, E)$ be a connected graph. For any k, where $1 < k \leq |E|$, there exists a $[k, 3k-3]$-partition of G.*

Proof. If $2 \leq k \leq |E| \leq 3k-3$, then E itself is an $[k, 3k-3]$-partition so there is nothing to prove. We will now prove the claim by strong induction on $|E|$. The induction hypothesis is:

> $\mathbf{P}(N)$: There exists a $[k, 3k-3]$-partition for any $G = (V, E)$ whenever $|E| \in [k, N]$.

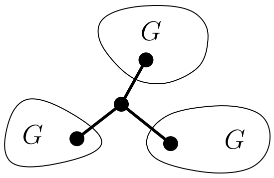
We claim that $\mathbf{P}(N)$ is true for all N. We know that $\mathbf{P}(3k-3)$ is true, so suppose that $\mathbf{P}(N)$ is true for some $N \geq 3k-3$, and consider $\mathbf{P}(N+1)$. Let $G = (V, E)$ be any graph with $|E| = N+1$. Since $|E| \geq 3k-2$, Lemma 2.2 implies that E can be

Algorithm 1 get_components(E, k)

// Returns a $[k, 3k-3]$-partition for the edge set E; Assume $|E| \geq k$;
if $|E| \leq 3k-3$ **then**
  return E;
Using DFS, compute $F \subset E$ and all its satellites; let α_1 be its largest satellite;
while $|\alpha_1| < k$ **do**
  Choose an edge $e \in F$ that is incident with a vertex in the subgraph induced by α_1;
  Choose an edge f in a satellite $\alpha_2 \neq \alpha_1$ which is incident to a node induced by F;
  if $F \setminus e$ is a connected edge set **then**
    $\alpha_1 \leftarrow \alpha_1 \cup e$; $F \leftarrow (F \setminus e) \cup f$;
  else if $F \setminus e$ is disconnected and $|\gamma_1| = |\gamma_2| = k-1$ **then**
    Label γ_1, γ_2 so that α_1 is connected to γ_1;
    return $\{\gamma_2 \cup e\} \cup$ get_components$(\alpha_1 \cup \gamma_1, k)$;
  else ($F \setminus e$ is disconnected into γ_1, γ_2 which are labeled so that $|\gamma_1| \geq k$)
    if $|\alpha_1 \cup \gamma_2| \geq k$ **then**
      return get_components$(\gamma_1, k) \cup$ get_components$(\alpha_1 \cup \gamma_2, k)$;
    else
      $\alpha_1 \leftarrow \alpha_1 \cup \gamma_2$;
      $F \leftarrow$ connected subset of γ_1 of size $2k-2$ that is adjacent to α_1;

decomposed into two disjoint connected edge sets E_1, E_2 with $k \leq |E_1| \leq |E_2| \leq N$. By the induction hypothesis, there exist $[k, 3k-3]$-partitions of E_1 and E_2. The union of these two partitions is a $[k, 3k-3]$-partition of E, concluding the proof. □

The following example proves that the result of Theorem 2.3 is tight. For a given k, let G be any connected graph with $k-2$ edges, and connect three such graphs in a wheel configuration as shown on the right. It is easy to see that the only decomposition in which every edge set has at least k edges is the entire graph itself, which has $3k-3$ edges.



The proof in Theorem 2.3 is constructive, based upon the construction in Lemma 2.2. In order to analyze the run time more easily, we convert the construction into the algorithmic format in Algorithm 1. We use the same notation that was used in the proof of Theorem 2.3. The DFS and the computation of the satellites is $O(E)$. The while loop executes at most k times, since $|\alpha_1|$ strictly increases in each execution (else the function calls itself and returns). In each execution, at most $O(E)$ work is done, and get_components can possibly be called on two smaller instances, both corresponding to graphs of size $\geq k$. Thus, letting $T(|E|, k)$ denote the worst case run time to obtain a $[k, 3k-3]$-partition for a graph with size $|E|$, we have that for some constant c,

$$T(|E|, k) \leq \max_{3k-3 \leq b \leq |E|-3k+3} \{T(b, k) + T(|E|-b, k)\} + ck|E|.$$

with $T(|E|, k) = 1$ for $|E| \leq 3k-3$. One can show by induction that $T(k, |E|) \leq \frac{3}{2}c|E|^2 = O(|E|^2)$, and hence the algorithm to compute the decomposition is polynomial in $|E|$.

2.2. The Region Graph. Consider a connected graph $G = (V, E)$, with n nodes. Take an $[\alpha, \beta]$-partition of G, which gives regions $R_1, R_2, \ldots, R_k$. Let the subgraphs induced by these regions have vertex sets $U_1, U_2 \ldots, U_k$. The *region graph* $G' = (V', E')$ has a vertex set $V' = \{r_1, r_2, \ldots, r_k\}$ where each vertex r_i corresponds to the region R_i of G. Two vertices r_i, r_j are adjacent in G' (that is, the edge (r_i, r_j) is in E') if and only if $U_i \cap U_j \neq \emptyset$ (that is, the corresponding regions are connected). An example of a region graph is given in Figure 1.1. Since each region consists of at least α and at most β edges, we immediately have that $|E|/\beta \leq |V'| \leq |E|/\alpha$. We proceed to show that G' is connected.

LEMMA 2.4. *Graph G' is connected.*

Proof. Let r_i, r_j be two nodes in V' corresponding to regions R_i, R_j in G. We show there is a path in G' from r_i to r_j. If R_i and R_j share a node then r_i and r_j are adjacent. Otherwise, let $e_i = (u_i, v_i) \in R_i$ and $e_j = (u_j, v_j) \in R_j$ be two edges in E. Since G is connected, there is a path in G from v_i to u_j. This path consists of edges $e_1, e_2, \ldots, e_k$ in regions $R_1, R_2, \ldots, R_k$, respectively. (Note that consecutive regions are not necessarily distinct.) Now consider the path $e_i, e_1, e_2, \ldots, e_k, e_j$ and the regions $R_i, R_1, R_2, \ldots, R_k, R_j$; since every two consecutive edges share a node, every two consecutive regions in this list are connected, which implies the existence of a walk from r_i to r_j in G'. Since r_i and r_j are arbitrary nodes, it follows that G' is connected. ∎

2.3. Paths on Region Graph. Let $\mathcal{P}$ denote a set of paths on G with edge congestion C, node congestion $\overline{C}$ and dilation D. Let $\{R_1, \ldots, R_k\}$ be an $[\alpha, \beta]$-partition of G into regions. Every edge in G belongs to exactly one region. Let $G' = (V', E')$ be the corresponding region graph. We define a mapping $f : E \rightarrow V'$ from the edges of G to the nodes of G' as follows.

For every $e \in E$, $f(e) = r_i$ if and only if $e \in R_i$.

Consider a path $p \in \mathcal{P}$, with $p = (e_1, e_2, \ldots, e_l)$. We define a mapping g which maps a path in G to a path in G' as follows.

For any path $p = (e_1, e_2, \ldots, e_l)$ in G, consider the walk in G' given by $w' = (f(e_1), f(e_2), \ldots, f(e_l))$. The path $g(p)$ denotes the walk w' that we obtain after removing all the cycles and repeated nodes in w', $g(p) = (f(e_{i_1}), f(e_{i_2}), \ldots, f(e_{i_k}))$ (see Figure 1.1).

We now transform the set of paths $\mathcal{P}$ of the original graph G into a set of paths $\mathcal{P}'$ on the region graph G' as follows.

$\mathcal{P}' = \{p'_1, p'_2, \ldots, p'_N\}$ where $p'_i = g(p_i)$, for every path $p_i \in \mathcal{P}$.

Let C', $\overline{C'}$ and D' denote the edge congestion, the node congestion and the dilation of the paths in $\mathcal{P}'$, respectively. For any set of paths, the edge congestion is trivially bounded by the node congestion; so $C' \leq \overline{C}'$. A path uses node r_i only if it contains edges in R_i. By construction, $|R_i| \leq \beta$, so the number of edges in $\mathcal{P}$ that use R_i is at most βC, thus, $\overline{C'} \leq \beta C$. Since $|g(p)| \leq |p|$ for any path p in G, we immediately have:

LEMMA 2.5 (Congestion and dilation in the region graph). *$C' \leq \overline{C'} \leq \beta C$; $D' \leq D$.*

2.4. Euler Cycles in Regions. Given an undirected graph $G = (V, E)$, we define the *directed representation* of G to be the graph $G^{dir} = (V, E^{dir})$ where each (undirected) edge $(u, v) \in E$ is replaced by two directed edges $(u, v), (v, u) \in E^{dir}$. Consider a graph decomposition of G into regions $R_1, R_2, \ldots, R_k$. We associate with each R_i a region R_i^{dir} in G^{dir}, where each edge in R_i is replaced by the two respective directed edges in R_i^{dir}. Take any node v induced by region R_i. Since every edge in R_i is replaced by two edges in opposite directions in R_i^{dir}, the in-degree of v is equal to its out-degree in R_i^{dir}.

An *Euler cycle* in a region R_i^{dir} is an edge-simple cycle that contains all the edges of R_i^{dir}. Since in R_i^{dir} the in-degree equals the out-degree of every node, R_i^{dir} has an Euler cycle. Let ψ_i denote an Euler cycle in R_i^{dir}. Let $\psi_i = (v_1, v_2, \ldots, v_1)$ be the sequence of nodes that ψ_i visits in R_i^{dir}. Since G and G^{dir} have the same set of nodes, ψ_i is mapped to a walk in R_i, when we follow the same sequence of nodes as in R_i^{dir}. We will refer to ψ_i as the *Euler cycle* of R_i as well.* In R_i, ψ_i will traverse the same edge twice, since the edge is traversed in two opposite directions in R_i^{dir}. Thus, in an $[\alpha, \beta]$-partition of G, every Euler cycle ψ_i satisfies $2\alpha \leq |\psi_i| \leq 2\beta$ (since $\alpha \leq |R_i| \leq \beta$).

3. Store-and-Forward Scheduling in G'. In graph G, the bufferless algorithm will emulate a store-and-forward algorithm which is applied in region graph G'. Here, we discuss the specifications for the store-and-forward algorithm. Let A denote such a store-and-forward algorithm. We will define the specifications of A. We will then give an instantiation of such an algorithm below (Algorithm A_1).

3.1. Specification of Algorithm A. Consider the batch problem $Q = (G, \Pi, \mathcal{S})$ and a set of paths $\mathcal{P}$ that satisfy Q. Let $\mathcal{P}'$ be the respective set of paths in G' (recall Section 2.3). The objective of the store-and-forward Algorithm A is to solve a *packet scheduling problem* $Q' = (G', \Pi, \mathcal{P}')$ in graph G', where the task is to send the packets Π along their respective set of paths $\mathcal{P}'$ in G'. Let γ denote the size of the *node-buffers* that Algorithm A uses. Specifically, each node has a node-buffer of size γ. A packet scheduling is *valid* whenever packets are not dropped (there are no buffer overflows). During any single time step in Algorithm A, a packet may perform one of four actions:

(i)	Be injected into the network at its source node.	[**Injection**]
(ii)	Move from its current node to a neighboring node.	[**Transfer**]
(iii)	Move to and be absorbed in its destination node.	[**Absorbtion**]
(iv)	Remain in the buffer of its current node.	[**Buffering**]

If a packet is received into a node's buffer at time t or was already buffered there during time step t, then at time $t+1$, it must either be transmitted along the next edge in its path or remain stored in the buffer. It is possible to divide any valid scheduling into a sequence of *phases*, so that each phase has the following three properties:

(i) Each phase is a time interval consisting of at least one time step.
(ii) During a phase, each packet traverses at most one edge in G'.
(iii) During a phase, a node receives at most γ packets (by transfer or injection).

Suppose that algorithm A produces a valid schedule. A trivial division of the execution of Algorithm A into phases that satisfies these three properties is to take each phase to be a single time step, since at every time step, any valid execution of Algorithm A must satisfy these three properties. (Property (iii) is satisfied for any valid scheduling because the size of the buffer is γ.) As we will show later, any candidate store-and-forward algorithm which produces valid schedules and satisfies these three properties may be used in our bufferless emulation algorithm to create a universal bufferless algorithm. In this case, we will say that the store-and-forward algorithm is *emulatable*. We now give a simple, randomized emulatable store-and-forward algorithm, which

*This is clearly an abuse of notation, since ψ_i is an Euler cycle of R_i^{dir}, not of R_i.

Algorithm 2 Store-and-Forward Algorithm A_1

1: Divide time into phases of length $\gamma = 6\log(n' + 2N)$ time steps.
2: **for** each packet π **do**
3: π will be injected at a phase ϕ_π, where ϕ_π is chosen uniformly and at random between 1 and $12\overline{C'}/\gamma$;
4: Packet π is injected at the first time step of phase ϕ_π;
5: Packet π follows its path traversing one edge per phase;

with high probability gives a valid schedule that satisfies the aforementioned three properties.

3.2. Store-and-Forward Algorithm A_1**.** We now give an instantiation of a universal, emulatable store-and-forward algorithm, which is actually the universal distributed algorithm of Leighton, Maggs, and Rao [35]. Here, we analyze the node-buffer requirements of the algorithm and bound its delivery time with respect to the node congestion, while originally in [35] the algorithm is analyzed with respect to edge-buffers and edge-congestion. The algorithm is randomized, uses node-buffers whose size γ is logarithmic with respect to the parameters of the scheduling problem, and has near-optimal delivery time. We refer to this algorithm as Algorithm A_1.

Let $Q' = (G', \Pi, \mathcal{P}')$ be a scheduling problem with path set $\mathcal{P}'$ on an arbitrary graph $G' = (V', E')$. Let $\overline{C'}$ be the node congestion and D' the dilation for the paths in $\mathcal{P}'$. Let N be the number of packets and $n' = |V'|$.

Algorithm 2 contains the details of the store-and-forward Algorithm A_1. The intuition behind this algorithm is that at most γ packets will be stored during a phase in any node. The packets will leave the node by the end of the phase. This is feasible, since the phase consists of γ time steps. However, we need to be a little careful to ensure that all γ packets that are leaving a node will find buffer space to enter their destination node. The detailed analysis of the algorithm follows below.

We will show that with high probability, Algorithm A_1 successfully delivers the packets, and at the same time it is emulatable. For Algorithm A_1 and phase ϕ, we define the following properties:

P1(ϕ): In phase ϕ every packet in the network successfully traverses one edge in its path.

P2(ϕ): No more than γ packets are buffered at any node during phase ϕ.

P3(ϕ): No more than $\gamma/2$ packets arrive at any node during phase ϕ.

P4(ϕ): No more than $\gamma/2$ packets remain at any node at the end of phase ϕ.

Note that *P3* is stronger than we need. We introduce property *P4* for technical convenience. Since the maximum injection phase is $12\overline{C'}/\gamma$ and the maximum path length is D', we have:

LEMMA 3.1. *If P1-P4 holds for* $12\overline{C'}/\gamma + D'$ *phases, then* $\Phi_{A_1}(Q') \leq 12\overline{C'}/\gamma + D'$*, and Algorithm* A_1 *is a valid algorithm for bufferless emulation (emulatable).*

Let $\Pr[\phi_0]$ be the probability that properties *P1-P4* hold for all phases $\phi \leq \phi_0$. $\Pr[0] = 1$ by default. We now give a lower bound for $\Pr[\phi_0 + 1]$ in terms of $\Pr[\phi_0]$.

LEMMA 3.2. *If P1-P4(*ϕ_0*) are true and P3(*$\phi_0 + 1$*) is true, then P1-P4(*$\phi_0 + 1$*) are true.*

Proof. If no more than $\gamma/2$ packets arrive at a node during phase $\phi_0 + 1$, then since *P4*(ϕ_0) is true, there are at most γ packets in the node during any time step of phase $\phi_0 + 1$, therefore *P2*($\phi_0 + 1$) is true. In the worst case all the at most $\gamma/2$ packets in the node at the end of phase ϕ_0 may leave sequentially on a single edge, requiring at most $\gamma/2$ time steps, which is less than the duration of the phase, so *P1*(ϕ_0+1) is true. The packets remaining in the node at the end of phase ϕ_0+1 are only those that entered, which is at most $\gamma/2$ packets, thus *P4*(ϕ_0+1) is true. ∎

By induction, we obtain the following corollary.

COROLLARY 3.3. *P1-P4(ϕ) are true for all $\phi \le \phi_0$ if and only if* P3*(ϕ) is true for all $\phi \le \phi_0$.*

Thus, $\Pr[\phi_0+1] = \Pr[\{$*P1-P4*(ϕ) are true for $\phi \le \phi_0\} \wedge \{$*P3*$(\phi_0+1)$ is true$\}]$. Noting that $\Pr[A \wedge B] = 1 - \Pr[\sim A \vee \sim B] \ge 1 - \Pr[\sim A] - \Pr[\sim B]$,

$$\Pr[\phi_0+1] \ge \Pr[\phi_0] - \Pr[\{P3(\phi_0+1) \text{ is false}\}]. \tag{3.1}$$

Consider a node v, and phase ϕ_0+1. Let q_π be the probability that packet π arrives at node v during phase ϕ_0+1 which can happen only if it is injected at a particular phase. Since the probability that it is injected at that particular phase is $\gamma/12\overline{C'}$, we conclude that $q_\pi \le \gamma/12\overline{C'}$ if π uses node v (at most $\overline{C'}$ such packets), and 0 otherwise. Let $X_i(v) = 1$ if packet π_i appears at node v at phase ϕ_0+1. $X_i(v)$ are independent random variables, whose sum is the number of packets that appear i n node v at phase ϕ_0+1. Let $X(v) = \sum_i X_i(v)$. $\mathbf{E}[X(v)] = \sum_i q_{\pi_i} \le \overline{C'} \cdot \gamma/12\overline{C'} = \gamma/12$. By applying a version of the Chernoff bound [38, Exercise 4.1] we obtain

$$\Pr[X(v) > \gamma/2] < 2^{-\gamma/2}.$$

Applying the union bound now gives that $\Pr[\max_v X(v) > \gamma/2] < n'2^{-\gamma/2}$, giving

LEMMA 3.4. $\Pr[\{$*P3*(ϕ_0+1) *is false*$\}] < n'2^{-\gamma/2}$.

Using (3.1), Lemma 3.4 and the fact that $\Pr[0]=1$, we get the following result by induction.

LEMMA 3.5. $\Pr[\phi_0] \ge 1 - \phi_0 n' 2^{-\gamma/2}$.

Since $n' < n' + 2N$, $12\overline{C'}/\gamma + D' < n' + 2N$ (because $\overline{C'} \le N$ and $D' \le n'$), and $2^{-\gamma/2} = (n'+2N)^{-3}$, by setting $\phi_0 = \Phi_{A_1}(Q')$ in Lemma 3.5, and using Lemma 3.1, we obtain the main result of this section:

THEOREM 3.6 (Delivery time of Algorithm A_1). *With probability at least* $1 - 1/(n'+2N)$*, Algorithm A_1 solves scheduling problem Q' in at most* $12\overline{C'}/\gamma + D'$ *phases, satisfying P1-P4 in each phase (algorithm A_1 is emulatable). The node-buffer size required is* $\gamma = 6\log(n'+2N)$.

4. Bufferless Emulation in G. Let $G = (V,E)$ be a connected graph with n nodes and let $\{R_1,\ldots,R_k\}$ be an $[\alpha,\beta]$-partition of G with corresponding region graph $G' = (V',E')$. Consider batch problem $Q = (G,\Pi,\mathcal{S})$ in G. Let $\mathcal{P}$ be a set of paths that satisfy Q. Let the corresponding path scheduling problem in G' be $Q' = (G',\Pi,\mathcal{P}')$.

Our goal is to design a bufferless algorithm to solve the batch problem Q, with delivery time $\widetilde{O}(C+D)$. In other words, the goal is to construct a set of paths $\widehat{\mathcal{P}}$ for which a bufferless and collision-free schedule exists with delivery time $\widetilde{O}(C+D)$. We implicitly obtain this set of paths through emulation of a store-and-forward algorithm. The store-and-forward algorithm solves the scheduling problem Q' in G'. with set of paths $\mathcal{P}'$; the set of paths $\mathcal{P}'$ were derived from $\mathcal{P}$. In solving Q in G, the bufferless algorithm will *emulate* Algorithm A in G' (in a step-by-step fashion). The set of paths $\widehat{\mathcal{P}}$ derived by the bufferless algorithm will depend on the set of paths in $\mathcal{P}'$, and hence also on the set of paths in $\mathcal{P}$. Thus, the preselected paths in $\mathcal{P}$ are crucial to the functioning of the bufferless algorithm, even though the paths used may deviate significantly from the preselected paths.

4.1. The Emulation. Assume that we have already constructed an emulatable, store-and-forward *Algorithm A* which solves the region graph scheduling problem $Q' = (G', \Pi, \mathcal{P}')$ using a buffer of size γ. Assume that $2\gamma \le |E|$ (we will deal with this assumption later in Section 5). We now discuss how to obtain a bufferless *Algorithm B* which will emulate the phases of Algorithm A, which may possibly be faster than emulating the individual time steps of Algorithm A. During a single phase of Algorithm A, a packet π performs one of four actions (in G'): injection; transfer; absorbtion; or, buffering. Algorithm B emulates Algorithm A phase for phase by emulating each of these actions that a packet can make. We continue with an informal description of algorithm B.

Algorithm B emulates the buffering of packets and their transfer from node to node using the $[\alpha, \beta]$-partition of G, where we set $\alpha = 2\gamma$ (by assumption, $\alpha = 2\gamma \le |E|$). By Theorem 2.3, we guarantee the existence of such an $[\alpha, \beta]$-partition by choosing $\beta = 6\gamma - 3$. Recall that we refer to nodes in the region graph by r_i and their corresponding region in the original graph by R_i.

- When in Algorithm A, a packet is buffered in a node r_i of G', Algorithm B emulates this by letting the packet circulate in the edges of region R_i in G.
- When in Algorithm A, a packet is transferred from node r_i to node r_j of G', in Algorithm B the packet is transferred from region R_i to region R_j in G; If the packet is absorbed by r_j, it will be absorbed by the appropriate node in R_j.
- If a packet is injected into the buffer of a node r_i by Algorithm A, then in Algorithm B, it will be injected into its injection node in R_i; From then on, it will continue to circulate in the region until it is transferred to the next region.

We now describe the details of the emulation.

4.1.1. Phases and Rounds. Let Φ denote the number of phases Algorithm A uses to deliver the packets in the region graph. In Algorithm B, time is divided into Φ phases. Each phase of Algorithm B emulates a phase of Algorithm A. In order to perform the emulation of a phase, Algorithm B further divides each phase into Ξ *rounds*, where Ξ will be specified later. The duration of each round is $Z = 4\beta^2 + 4\beta$ time steps. (Recall that $\beta = 6\gamma - 3$.) Thus, the bufferless algorithm runs for $\Phi \cdot \Xi \cdot Z$ time steps in total.

For the duration of an entire round, a region is either in the *sending* or the *receiving* mode – we say that the region is sending or receiving. In the emulation, when a packet has to be transferred from region R_i to the region R_j, R_i must be sending and R_j receiving. We will show how to guarantee that for any pair of adjacent nodes $r_i, r_j \in V'$, there is a round in every phase in which region R_i is sending and R_j is receiving (and vice-versa).

In order to determine if a region is sending or receiving, we first obtain a vertex coloring of G'. Let $\boldsymbol{\chi} : V' \mapsto [0, n']$ be a *valid* vertex coloring of G', where $\boldsymbol{\chi}(r)$ is the color assigned to node $r \in V'$, and no two nodes have the same color. Let $\chi = \max_i \boldsymbol{\chi}(r_i)$ denote the maximum color used in the the vertex coloring. A valid coloring can be obtained by a simple greedy algorithm where the maximum color is bounded by the maximum node degree. Since the maximum node degree is bounded by n', where $n' = |V'|$, we have that $\chi \le n'$.

We define the color of a region R_i as the color assigned to the corresponding node r_i. Let $\boldsymbol{\delta}_i$ be the binary representation of $\boldsymbol{\chi}(r_i)$. Let σ denote the number of bits in χ, $\sigma = \lceil \log \chi \rceil \le \lceil \log n' \rceil$. By pre-padding with zeros, we assume that every color $\boldsymbol{\delta}_i$

has σ bits. We define the *mode parameter* $\mathbf{x}_i$ for region R_i to be the 2σ long binary vector $\bar{\boldsymbol{\delta}}_i\boldsymbol{\delta}_i$, where $\bar{\boldsymbol{\delta}}_i$ is the binary complement of $\boldsymbol{\delta}_i$. For $1 \leq k \leq 2\sigma$, we denote the k-th bit of $\mathbf{x}_i$ by $\mathbf{x}_i(k)$.

We set the number of rounds in a phase to be $\Xi = 2\sigma \leq 2\lceil \log n' \rceil$; thus, each phase in Algorithm B consists of the 2σ rounds, $\omega_1, \omega_2, \ldots, \omega_{2\sigma}$. During round ω_k, if $\mathbf{x}_i(k) = 0$ then region R_i is sending, otherwise, if $\mathbf{x}_i(k) = 1$, then region R_i is receiving. Our assignment of colors ensures that during every phase, any region R_i may send a packet to any neighboring region R_j (i.e. R_i will be sending and R_j receiving), and similarly it may receive a packet from any neighboring region R_j (i.e. R_j will be sending and R_i receiving).

Lemma 4.1. *If R_i and R_j are adjacent, then during every phase, there is at least one round ω_s (ω_r) in which R_i is sending (receiving)* and *R_j is receiving (sending).*

Proof. Since $\boldsymbol{\chi}$ is a valid coloring, and R_i and R_j are adjacent, $\boldsymbol{\delta}_i$ and $\boldsymbol{\delta}_j$ must differ at some bit. Suppose they differ in the k^{th} bit, $1 \leq k \leq \sigma$. Thus, rounds k and $k+\sigma$ satisfy the requirements, since $\overline{\mathbf{x}_i(k+\sigma)} = \mathbf{x}_i(k) = \overline{\mathbf{x}_j(k)} = \mathbf{x}_j(k+\sigma)$. ☐

The fundamental operation that is needed for the emulation by Algorithm B is packet circulation within a region.

4.1.2. Packet Circulation. Packet circulation is a basic function for the emulation. During packet circulation, a packet π repeatedly follows the Euler cycle ψ_i of the region R_i that it is in: at each time step, packet π follows the next edge in ψ_i; when π reaches the end of the Euler cycle it continues from the beginning of the cycle, and so on. At the time step in which packet π traverses an edge $e \in \psi_i$, we say that e is the *current* edge of π.

At each round of a phase, a region is either sending or receiving. The speed at which a packet circulates in its region depends on whether the region is sending or receiving:

- If the region is receiving, then the packet follows the Euler cycle in the normal fashion (one link per time step).
- If the region is sending, then the packet moves at an effectively slower speed as follows. At time step 0 (the beginning of the round), suppose that π is at node u with current edge $e = (u, v) \in \psi_i$. At time step 0, packet π follows its current edge (u, v) and at time step 1, π appears in node v. At time step 1, suppose that its new current edge in ψ_i is (v, w); the packet *does not* follow its new current edge in ψ_i, but instead it follows edge (v, u) from v back to u, and thus at time step 2, it appears back in node u. Thus after two time steps, the packet has effectively not moved. We call such an operation an *oscillation*, and we say that packet π oscillates on its current edge in the Euler cycle. The time period of the oscillation is 2 time steps, The packet continues in this fashion for subsequent time steps, so at even time steps $t = 2i$, it appears in node u, and at odd time steps $t = 2i + 1$ it appears in node v, for $i \geq 0$. The packet performs β such oscillations on its current edge e; so, after 2β time steps, the packet appears at u and follows edge e for the last time. At time step $T_s = 2\beta + 1$, the packet is now at v and at this point it stops oscillating on edge e and begins oscillating on its new current edge $(v, w) \in \psi_i$. Thus, after T_s time steps, the packet advances by one edge in the Euler cycle of ψ_i. Consequently, since $|\psi_i| \leq 2\beta$, after $2\beta T_s = 4\beta^2 + 2\beta$ time steps, a packet circulating in region R_i has oscillated at least once on every edge of ψ_i.

From the above description of the packet movement in a sending region, we obtain:

LEMMA 4.2. *After $4\beta^2 + 2\beta < Z$ time steps, a packet circulating in a sending region R_i has oscillated at least once on every edge in ψ_i.*

Suppose that the directed edge $e = (u, v) \in \psi_i$, is an edge in the Euler cycle of a receiving region R_i. If at time step t, no packet has edge e as its current edge, then we say that e is *empty.* At each time step, we say that an empty edge is associated with an *empty slot.* Empty slots are similar to packets in that they too circulate – as the packets in a receiving region circulate (forwards) in ψ_i, the empty slots circulate (backwards) in ψ_i at the same rate. They continue to circulate until some packet occupies the empty edge.

Once a packet enters the network, its default status is to be circulating in the region it is in. Packets enter a region either through injection or packet transfer. We discuss how these steps are emulated by bufferless Algorithm B. In particular, whenever a packet enters a region, it must not interfere with packets that are already circulating in the region.

4.1.3. Emulation of Injection. Suppose that in phase ϕ, Algorithm A injects packet π into node r_i. Let p be the path of π in G, e the first edge in this path, and u the injection node. Since Algorithm A injects the packet into node r_i of G', we know that e and u are in R_i. Algorithm B will inject π into R_i in phase ϕ during the last round of the phase in which R_i is receiving. Algorithm B will inject π into an empty edge in the Euler cycle ψ_i, i.e., one which is not a current edge of any packet circulating in R_i. In this way, we guarantee that the injection of π will not interfere with any currently circulating packets. After injection, π will continue to circulate in R_i until the end of phase ϕ. All that remains is to show that it is possible to inject π so that it does not interfere with any circulating packets. We will say that a region is *ready for phase ϕ* of bufferless Algorithm B if at the begining of phase ϕ (end of phase $\phi - 1$) there are at most γ packets circulating in the region. By default, all regions are ready for the first phase (since there are no packets in the network).

Remember that $2\alpha \leq |\psi_i| \leq 2\beta$, so $4\gamma \leq |\psi_i|$. Suppose that region R_i is ready for phase ϕ of Algorithm B. Then, by definition of a phase in Algorithm A, at most γ packets will need to enter region R_i during phase ϕ, i.e., at most $\gamma - 1$ packets other than π need to enter. This means that only at most $2\gamma - 1$ of the edges in ψ are current edges of packets, the remaining $|\psi_i| - (2\gamma - 1) \geq 2\alpha - (2\gamma - 1) = 2\gamma + 1$ edges are empty slots that continually circulate backwards one edge at a time during the round. An empty time slot is *available* for π if it will not be occupied by any packet other than π during its backward circulation. Thus, there are at least $2\gamma + 1$ available empty slots for π in region R_i. For any given edge e, each of these available empty slots will be at e once every $|\psi_i| \leq 2\beta < Z$ time steps.

Let $e = (u, v) \in \psi_i$ be the first edge of packet π in R_i. Packet π is injected into the network and e becomes its current edge at the first time step when e becomes empty. Since π is injected into an empty edge, it does not interfere with any packets already circulating in R_i. There are at least $2\gamma + 1$ available empty slots, so we know that e becomes empty at least $2\gamma + 1$ times during the round. Thus, even if all (at most γ) packets entering R_i during this phase are through injection at node u with the *same* first edge $e = (u, v)$ in their path, all of these packets can be injected into circulation in R_i in just this one round.

LEMMA 4.3. *Suppose that packet π with first edge $e \in R_i$ is injected into node r_i during phase ϕ of Algorithm A. If R_i is ready for phase ϕ of Algorithm B, then packet π can be injected into R_i without interfering with any already circulating packets.*

4.1.4. Emulation of Packet Transfer. Suppose that in phase ϕ of Algorithm A, packet π moves from node r_i to node r_j. Assume that at the beginning of phase ϕ in Algorithm B, packet π is circulating in region R_i, and that region R_j is ready for phase ϕ of Algorithm B. During phase ϕ in Algorithm B, π will move from R_i to R_j as follows. Packet π will circulate in R_i until the first round ω of phase ϕ in which R_i is sending and R_j is receiving. (The existence of such a round is guaranteed by Lemma 4.1.)

Since r_i and r_j are adjacent in G', there exists a node u which is common to R_i and R_j. Since node u is in R_i, there exists an edge $e_i = (u_i, u) \in \psi_i$ on the Euler cycle of R_i. Similarly, there exists an edge $e_j = (u, u_j) \in \psi_j$ on the Euler cycle of R_j. During round ω, packet π circulates (in slow mode) in region R_i along the Euler cycle ψ_i. At some particular slow time step τ of the round, the current edge of π will be e_i. During the next $T_s = 2\beta+1$ time steps, π oscillates on edge e_i, and will appear at the common node u at the $\beta+1$ times $\tau+1, \tau+3, \ldots, \tau+2\beta+1$. If at any of these times, the edge $e_j \in \psi_j$ is an empty slot, i.e., not the current edge of any packet circulating (in normal mode) in R_j, then π switches from oscillation on edge e_i, making e_j its new current edge. Packet π now continues to circulate in R_j at normal speed. Since π enters R_j on an empty edge, it will not interfere with any packets already circulating in R_j. Note that π will have completed a full circuit on its Euler path ψ_i in at most $4\beta^2 + 2\beta$ time steps, thus, it will have completed its oscillations on edge e_i within the first $4\beta^2 + 2\beta$ time steps of the round. Thus, π will enter R_j within the first $4\beta^2 + 2\beta$ time steps of round ω, provided that it found an empty edge on which to enter.

We now show that during round ω, π will indeed find an empty edge on which to move into R_j. Specifically, for at least one of the time steps $\tau + 1, \tau + 3, \ldots, \tau + 2\beta + 1$, the edge $e_j \in \psi_j$ will be an empty slot. Remember that empty slots circulate backwards in R_j at the rate of one edge per time-step. Thus, *every* available empty slot will pass e_j at least once during *any* consecutive 2β time steps. By the arguments in Section 4.1.3, we know that there are at least $2\gamma + 1$ such available empty slots. Therefore, edge e_j will become an empty slot at least once in the $2\beta + 1$ consecutive time steps $\tau+1, \tau+2, \tau+3, \ldots, \tau+2\beta+1$. However, due to the nature of packet π's oscillation on edge e_i, packet π will only be able to use an empty slot if the slot passes e_j at time step $\tau + k$ for some *odd* $k \in [1, 2\beta + 1]$. To show that such a situation is guaranteed to occur, we need to show that there is at least one pair of *consecutive* available empty slots.

LEMMA 4.4. *Suppose R_j is ready for phase ϕ of Algorithm B. There is at least one pair of consecutive empty slots that is available for π.*

Proof. Say that a slot is *booked* if it is not available for π. Let $\Gamma \leq 2\gamma - 1$ be the number of booked slots. The number of available empty slots is $|\psi_j| - \Gamma$. Since $|\psi_j| \geq 4\gamma$, we have that $\Gamma < |\psi_j|/2$. Suppose there is no pair of consecutive available slots for π. For every available slot, the next slot must therefore be booked, hence the number of available slots is at most $|\psi_j|/2$. Thus, $|\psi_j|$, the total number of slots (booked plus available), is at most $\Gamma + |\psi_j|/2 < |\psi_j|$, a contradiction. ☐

Let the two consecutive available slots implied by Lemma 4.4 be c_1 and c_2. Suppose that c_1 passes e_j first at time step $\tau + k_1$ for some $k_1 \in [0, 2\beta - 1]$. If k_1 is odd, then this empty slot can be used by π. If not, then c_2 passes e_j at time step $\tau + k_1 + 1$, where $k_1 + 1 \in [1, 2\beta]$ is odd, and so can be used by π. We have therefore shown:

LEMMA 4.5. *Suppose that packet π is transferred from r_i to r_j in phase ϕ of Algorithm A. In Algorithm B, suppose that R_j is ready for phase ϕ, and that at the begining of phase ϕ, π is circulating in region R_i. Then, π can be transferred from* *region R_i to R_j during the first $4\beta^2+2\beta$ time steps of a round in phase ϕ of Algorithm B.*

Note that since the packet transfers over into an empty edge, it does not interfere with any packets that were already circulating.

4.1.5. Emulation of Absorbtion. Suppose that packet π moves from node r_i to its destination node r_j in phase ϕ in store-and-forward Algorithm A (and is absorbed). Assume that R_j is ready for phase ϕ of Algorithm B and that π is circulating in R_i at the beginning of phase ϕ. We use the packet transfer emulation (Section 4.1.4) to first move the packet from region R_i to R_j in phase ϕ. By Lemma 4.5, this can be done in the first $4\beta^2 + 2\beta$ time steps of a round of phase ϕ in which R_i is sending and R_j receiving. The packet then circulates in R_j at normal speed until it reaches its destination node, at which point it is absorbed. Since the packet completes the Euler cycle for R_j in at most 2β time steps, the number of time steps to be transferred, circulate and be absorbed is at most $4\beta^2 + 4\beta \leq Z$, and this implies:

LEMMA 4.6. *Suppose that packet π is transferred from r_i to r_j where it is absorbed in phase ϕ of Algorithm A. In Algorithm B, suppose that R_j is ready for phase ϕ, and that at the beginning of phase ϕ, π is circulating in region R_i. Then, π can be transferred from region R_i to R_j and absorbed during a single round of phase ϕ in Algorithm B.*

4.1.6. Emulation of Buffering. Suppose that packet π is buffered at node r_i during phase ϕ of Algorithm A. Assume that in Algorithm B, packet π is already circulating in region R_i. Packet π will then continue to circulate in R_i uninterrupted through the entire phase ϕ. This is certainly possible unless some new packets entered the region (by transfer or injection) into the current edge of π. As we have already shown, injected or transferred packets do not interfere with already circulating packets, since they always enter on empty edges. We have the following lemma.

LEMMA 4.7. *If packet π is circulating in R_i at the end of phase $\phi-1$ of Algorithm B, and in phase ϕ of Algorithm A, π is buffered at r_i, then in phase ϕ of Algorithm B, it can be buffered in R_i using circulation.*

4.2. Analysis of Emulation. First, we prove that Algorithm B correctly emulates Algorithm A. We then analyze the delivery time of Algorithm B in G in terms of the delivery time of Algorithm A in G'.

4.2.1. Correctness. Assume that $\alpha = 2\gamma \leq |E|$ in order to guarantee the existence of the $[\alpha, \beta]$-partition. Algorithm B correctly emulates algorithm A phase by phase if, at the end of every phase ϕ, the following two statements hold:

(i) in Algorithm A, packet π is in node r_i iff in Algorithm B it is circulating in region R_i

(ii) in algorithm A packet π is injected (absorbed) at node r_i, if and only if in Algorithm B packet π is injected (absorbed) into region R_i.

We now prove by induction on the phase number ϕ that Algorithm B correctly emulates Algorithm A. Observe that when $\phi = 1$, Algorithm A can only inject packets into nodes. The conditions of Lemma 4.3 are satisfied, and since at most γ packets are injected into a node in G', Algorithm B can successfully inject these packets into the corresponding regions. Suppose that Algorithm B correctly emulates Algorithm A up to phase $\phi_0 \geq 1$. At the end of phase ϕ_0, there are at most γ packets circulating in any region R_i since every packet π in node r_i in the execution of Algorithm A is in region R_i in the execution of Algorithm B, and during the last time step of phase ϕ, Algorithm A can only be buffering at most γ packets (all other packets have left to adjacent nodes). Thus, the conditions of Lemmas 4.3, 4.5, 4.6, and 4.7, are satisfied for every packet π. Every action that π could take in phase $\phi_0 + 1$ of Algorithm A can now be emulated in phase $\phi_0 + 1$ of Algorithm B. By induction, we now have:

THEOREM 4.8 (Correctness of Emulation). *Algorithm B correctly emulates in G every phase in the execution of Algorithm A in G'. Hence, Algorithm B solves the batch problem Q without outgoing edge-buffers, and implicitly constructs the paths $\widehat{\mathcal{P}}$ and a collision-free schedule in them.*

4.2.2. Bufferless Delivery Time. Let $\mathcal{T}_B(Q)$ be the delivery time for bufferless Algorithm B to solve the batch problem Q (using initial paths $\mathcal{P}$), and let $\Phi_A(Q')$ be the number of phases required by Algorithm A to solve scheduling problem Q' (corresponding to the batch problem Q) on the region graph G'. Since Algorithm B emulates Algorithm A phase for phase, Algorithm B uses the same number of phases as Algorithm B, i.e., $\Phi_B(Q) = \Phi_A(Q')$. The delivery time is given by

$$\begin{aligned}\mathcal{T}_B(Q) &\leq \Xi \cdot Z \cdot \Phi_B(Q),\\ &\leq 576\gamma^2 \lceil \log \chi \rceil \cdot \Phi_A(Q'),\end{aligned}$$

where we have used $Z = 4\beta^2 + 4\beta \leq 8\beta^2$, $\beta = 6\gamma - 3 \leq 6\gamma$ and $\Xi = 2\sigma = 2\lceil \log \chi \rceil$ (χ is the chromatic number of the region graph G').

THEOREM 4.9 (Bufferless delivery time). $\mathcal{T}_B(Q) \leq c \cdot \Phi_A(Q') \cdot \gamma^2 \cdot \log \chi$ *for some constant c.*

Since $\chi \leq n' \leq |E|/\alpha = O(n^2)$, we have that $\mathcal{T}_B(Q) = O(\Phi_A(Q') \cdot \gamma^2 \cdot \log n)$.

Recap.. The bufferless algorithm solves the batch problem by deterministically emulating a store-and-forward algorithm on a corresponding scheduling problem in the region graph. Buffering is replaced by packet circulation, and the cost of the emulation is $O(\gamma^2 \cdot \log n)$, where γ is the node-buffer size required by the store-and-forward algorithm. Any store-and-forward algorithm that satisfies the two required properties for emulation (Section 3) will give a valid bufferless algorithm. The more efficient the store-and-forward algorithm is (in terms of phases) and the smaller the buffer size used, the more efficient the bufferless algorithm will be.

5. A Universal Bufferless Algorithm. By combining the results in Sections 3 and 4 (specifically Theorems 3.6 and 4.9), we will obtain a specific randomized universal bufferless algorithm which we denote *Algorithm* B_1. Algorithm B_1 emulates the store-and-forward Algorithm A_1. The buffer size required by algorithm A_1 is $\gamma \geq 6\log(n' + 2N)$. Since $n' \leq |E|/\alpha \leq |E|$, we can set $\gamma = 6\log(|E| + 2N)$. In order to apply the bufferless emulation algorithm, we need an $[\alpha, \beta]$-partition, where $\alpha = 2\gamma$ and $\beta = 6\gamma - 3$. Since $\alpha \leq |E|$, we must have that $2\gamma \leq |E|$. Substituting the expression for γ, we find that $2N \leq 2^{|E|/12} - |E|$. Thus, for the case where $2N > 2^{|E|/12} - |E|$ we need to apply a different approach. We examine these two cases separately.

5.1. The Case $2N \leq 2^{|E|/12} - |E|$. In this case, we can apply the bufferless emulation with $\gamma = 6\log(|E| + 2N)$. Note that γ is independent of the size of G'. Combining Theorems 3.6 and 4.9, we obtain

$$\begin{aligned}\mathcal{T}_{B_1}(Q) &\leq c \cdot \Phi_{A_1}(Q') \cdot \gamma^2 \log \chi,\\ &\leq c \cdot \left(12\frac{\overline{C'}}{\gamma} + D'\right) \cdot \log^2(|E| + 2N) \cdot \log \chi,\\ &\leq c \cdot (C + D) \cdot \log^2(|E| + 2N) \cdot \log |E|,\end{aligned}$$

where c represents a generic constant, not necessarily the same from line to line. The last inequality follows by using Lemma 2.5 and the facts that $\beta \le 6\gamma$ and $\chi \le n' \le |E|/\alpha \le |E|$. Since the randomized store-and-forward Algorithm A_1 succeeds with probability at least $1 - 1/(n' + 2N)$, the bufferless Algorithm B_1 has the same probability of success (i.e. it correctly sends the packets with the advertised delivery time with the same probability). Since $n' \ge |E|/\beta$, $\beta = 6\gamma - 3$, $\gamma = 6\log(|E| + 2N)$, and $|E| \le n^2$, we have that the probability of success is at least

$$1 - \frac{1}{n' + 2N} \ge 1 - \frac{1}{\frac{|E|}{36\cdot\log(|E|+2N)-3} + 2N} = 1 - O((n+N)^{-\lambda}),$$

for some constant $\lambda > 0$.

We now consider how the bufferless emulation of the store-and-forward algorithm can be performed in a distributed manner by the nodes of network G; that is, packet forwarding decisions can be made locally at each node. In order to do so, we need to assume that every node in G knows the following before the algorithm starts: the network topology G, the value of congestion C, and the number of packets N (such assumptions are commonly made in distributed bufferless routing algorithms [17, 34]). Based on these parameters, each node in G can compute the parameters which are necessary for the emulation: such as the structure of G', the buffer size γ, the duration of phases, etc.. In this way, the nodes in G know what kind of actions to perform at each time step in order to emulate the actions of the store-and-forward algorithm A_1. Note that that a node does need to have a-priori information about the packets with origin at other nodes.

Alternatively, instead of knowing G, each node could have been supplied a-priori information about G'. In particular, each node needs to have information about the regions of G' that it participates to. However, G' depends on N, since each region has $O(\log(|E| + N))$ edges. Different batch problems may have different values of N and use different graph G'. If we write $2^y \le N < 2^{y+1}$, where $0 \le y \le |E|/12$, we observe that there are in total $\Theta(|E|)$ different graphs G' that we could use for the emulation, one graph for each range $[2^y, 2^{y+1})$ of the value N. Each node in G could have been informed about all these possible graphs of G', and choose an appropriate one for the current value of N.

5.2. The Case $2N > 2^{|E|/12} - |E|$. In this case, we send the N packets of batch problem Q to their destinations one after the other along their pre-specified paths in G. Each packet takes time at most D to be delivered to its destination; thus, the total delivery time to send all the packets is at most DN. By the pigeonhole principle, $C \ge N/|E|$, and thus $C > (2^{|E|/12} - |E|)/2|E|$. Since $|E| = O(\log N)$ and $D \le |E|$, the delivery time is $ND \le CD|E| = O(C\log^2 N)$.

This simple algorithm can be converted to a distributed algorithm, where nodes make local decisions about packets, using packet priorities. There are two packet priorities, 0 and 1; in collisions, packets with priority 1 win, while packets with priority 0 are deflected. The details are given below. The algorithm proceeds in phases of duration $|E| = O(\log N)$.

i. At the beginning of the first phase, for each out-going edge e, a node injects at most one packet (if it has one) with e being the first edge in the path of the packet. All packets start with priority 1.
ii. For the duration of the phase, priority 1 packets always try to move to their destinations along their path, unless there is a collision. In a collision, priority

1 packets have precedence over priority 0 packets. If multiple priority 1 packets collide, then one of them (arbitrarily) wins the collision and all the other priority 1 packets involved in the collision drop to priority 0.

iii. Once a packet becomes priority 0, it randomly follows any available edge from its current node toward its destination in hot-potato style. If a packet of priority 0 happens to arrive at its destination during a phase, then it is absorbed.

iv. At the end of the phase, each remaining packet resets its path to be the shortest path from its current node to its destination.

v. For each subsequent phase, all nodes inject at most one packet per out-going edge as at the beginning of the first phase, with one exception: they do not inject a packet with initial edge e if some other packet is already at the node from the previous phase and has e as the next edge in its new (shortest) path to its destination. All packets (newly injected and from previous phase) begin the phase with priority 1.

vi. The entire process repeats for N phases.

If at the beginning of the phase there are packets in the network (newly injected of from previous phase), then all of them have priority 1. During the phase, at least one packet will retain priority 1, since in collisions involving packets with priority 1, at least one priority 1 packet survives. Further, in every time step, a priority 1 packet (unless it drops to priority 0) moves one edge closer to its destination. Since any path is no longer than the number of edges in the network, after $D \leq |E|$ time steps, all the priority 1 packets that did not drop to priority 0 (at least 1 of them exists) have been absorbed at their respective destinations. Thus, at least one packet is absorbed in each phase that starts with at least one packet. Note that if nodes have packets to inject, then at the beginning of the phase there is at least one packet in the network (newly injected or from the previous phase). Therefore, at most N phases are needed. Since each phase has $|E|$ time steps, the total delivery time is at most $N|E| \leq C|E|^2 = O(C \log^2 N)$.

5.3. Main Result. To wrap up, when $2N \leq 2^{|E|/12} - |E|$, we use bufferless emulation which with high probability obtains a delivery time of $O((C+D)\log^2(|E|+2N)\log|E|)$. Otherwise, we use the the simple brute-force algorithm of sending the packets one by one, which has delivery time $O(C \log^2 N)$. Combining these two results and using the fact that $|E| = O(n^2)$, we have a universal bufferless algorithm B_1 which with high probability has near optimal delivery time.

THEOREM 5.1 (Delivery time of Algorithm B_1). *Given paths $\mathcal{P}$ that satisfy batch problem Q, bufferless algorithm B_1 delivers the packets in time $\mathcal{T}_{B_1}(Q) = O((C+D) \cdot \log^3(n+N))$, with probability $1 - O((n+N)^{-\lambda})$, for some constant $\lambda > 0$.*

When choosing optimal initial paths, Theorem 5.1 establishes our main result, Theorem 1.1. Furthermore, using the distributed version for each case $2N \leq 2^{|E|/12} - |E|$ and $2N > 2^{|E|/12} - |E|$, we obtain the distributed version of B_1.

6. Discussion. Our main goal has been to establish the existence of universal, near optimal (to within poly-log factors) bufferless communication algorithms. Our proof of this fact has been *constructive*, using a bufferless emulation technique to emulate a store-and-forward algorithm on a batch scheduling problem related to the original batch packet problem. The heart of the emulation is to replace buffering with packet circulation. The algorithm we have given is distributed modulo the need for nodes to know C and N. Note that in Theorem 5.1, it is crucial to allow the paths to deviate from the pre-selected paths, otherwise it is possible to construct problems for which the optimal bufferless routing time is at least a $\sqrt{N}$ factor from optimal [22].

We briefly discuss some interesting directions for future work. The most natural question is whether some of the poly-log factors can be removed. Two of the poly-log factors arise purely from the bufferless emulation, and our guess is that these poly-log factors may not be necessary. Specifically, for a given batch problem Q with N packets on a network G, we can define the *Q-bufferless efficiency* $\rho_B(Q; N, G)$ as the ratio between the smallest possible delivery time of a bufferless algorithm for Q and the smallest possible delivery time of a store-and-forward algorithm for Q. The *bufferless efficiency* $\rho_B(N, G)$ is the maximum possible value of the Q-bufferless efficiency over all batch problems, $\rho_B(N, G) = \sup_Q \rho_B(Q; N, G)$. Our result shows that the bufferless efficiency is poly-logarithmicaly bounded, $\rho_B(N, G) = O(\log^3(n + N))$, where n is the size of G. An interesting problem is to determine tighter asymptotic upper bounds as well as lower bounds for the bufferless efficiency for specific as well as arbitrary networks (for example, it is shown in [21] that the bufferless efficiency for leveled networks is only $O(\log(n + N)))$.

A related question is whether special purpose store-and-forward scheduling algorithms can be used with our emulation technique to obtain *optimal* bufferless delivery time on specific classes of networks. If the region graph can be efficiently colored (for example fixed degree region graphs) and if the node-buffering requirement on such region graphs is some constant, then the resulting bufferless emulation only adds an additional constant factor to the delivery time. A good candidate for such a result is the mesh network, since a natural decomposition into regions would yield another mesh-like network, with good properties.

A slightly different line of enquiry is to determine whether the algorithm we have given can be made dynamic (i.e., not requiring *a priori* knowledge of C and N), in addition to being distributed. Such a result would show the existence of near-optimal bufferless algorithms for dynamic packet problems.

Acknowledgements. We are indebted to the referees of this journal for their valuable and insightful comments.

REFERENCES

[1] M. Adler, S. Khanna, R. Rajaraman, and Adi Rosen, *Time-constrained scheduling of weighted packets on trees and meshes*, in Proceedings of 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA), 1999.

[2] M. Adler, A. L. Rosenberg, R. K. Sitaraman, and W. Unger, *Scheduling time-constrained communication in linear networks*, in Proceedings of 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA), 1998.

[3] N. Alon, F.R.K. Chung, and R.L.Graham, *Routing permutations on graphs via matching*, SIAM Journal on Discrete Mathematics, 7 (1994), pp. 513–530.

[4] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup, *Direct routing on trees*, in Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 98), 1998, pp. 342–349.

[5] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts, *On-line routing of virtual circuits with applications to load balancing and machine scheduling*, Journal of the ACM, 44 (1997), pp. 486–504.

[6] B. Awerbuch and Y. Azar, *Local optimization of global objectives: competitive distributed deadlock resolution and resource allocation*, in Proceedings of 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, 1994, pp. 240–249.

[7] A. Bar-Noy, P. Raghavan, B. Schieber, and H. Tamaki, *Fast deflection routing for packets and worms*, in Proceedings of the Twelth Annual ACM Symposium on Principles of Distributed Computing, Ithaca, New York, USA, Aug. 1993, pp. 75–86.

[8] P. Baran, *On distributed communications networks*, IEEE Transactions on Communications, (1964), pp. 1–9.

[9] I. Ben-Aroya, T. Eilam, and A. Schuster, *Greedy hot-potato routing on the two-dimensional mesh*, Distributed Computing, 9 (1995), pp. 3–19.

[10] A. Ben-Dor, S. Halevi, and A. Schuster, *Potential function analysis of greedy hot-potato routing*, Theory of Computing Systems, 31 (1998), pp. 41–61.

[11] P. Berenbrink and C. Scheideler, *Locally efficient on-line strategies for routing packets along fixed paths*, in Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, N.Y., Jan. 17–19 1999, ACM-SIAM, pp. 112–121.

[12] S. N. Bhatt, G. Bilardi, G. Pucci, A. G. Ranade, A. L. Rosenberg, and E. J. Schwabe, *On bufferless routing of variable-length message in leveled networks*, IEEE Trans. Comput., 45 (1996), pp. 714–729.

[13] A. Borodin and J. E. Hopcroft, *Routing, merging, and sorting on parallel models of computation*, Journal of Computer and System Sciences, 30 (1985), pp. 130–145.

[14] A. Borodin, Y. Rabani, and B. Schieber, *Deterministic many-to-many hot potato routing*, IEEE Transactions on Parallel and Distributed Systems, 8 (1997), pp. 587–596.

[15] J. T. Brassil and R. L. Cruz, *Bounds on maximum delay in networks with deflection routing*, IEEE Transactions on Parallel and Distributed Systems, 6 (1995), pp. 724–732.

[16] A. Broder and E. Upfal, *Dynamic deflection routing on arrays*, in Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, May 1996, pp. 348–358.

[17] C. Busch, *$\widetilde{O}$(congestion + dilation) hot-potato routing on leveled networks*, Theory of Computing Systems, 37 (2004), pp. 371–396.

[18] C. Busch, M. Herlihy, and R. Wattenhofer, *Hard-potato routing*, in Proceedings of the 32nd Annual ACM Symposium on Theory of Computing, May 2000, pp. 278–285.

[19] ———, *Randomized greedy hot-potato routing*, in Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, Jan. 2000, pp. 458–466.

[20] ———, *Routing without flow control*, in Proceedings of the Thirteenth ACM Symposium on Parallel Algorithms and Architectures, July 2001, pp. 11–20.

[21] C. Busch, S. Kelkar, and M. Magdon-Ismail, *Efficient bufferless routing on leveled networks*, in Proceedings of the 11th International Conference on Parallel and Distributed Computing (Euro-par 2005), LNCS 3648, Lisboa, Portugal, Augoust-September 2005, pp. 931–940.

[22] C. Busch, M. Magdon-Ismail, M. Mavronicolas, and P. Spirakis, *Direct routing: Algorithms and Complexity*, Algorithmica, 45 (2006), pp. 45–68.

[23] C. Busch, M. Magdon-Ismail, M. Mavronicolas, and R. Wattenhofer, *Near-optimal hot potato routing on trees*, in Proceedings of Euro-Par 2004, LNCS 3149, Pisa, Italy, August-September 2004, pp. 820–827.

[24] R. Cypher, F. Meyer auf der Heide, C. Scheideler, and B. Vöcking, *Universal algorithms for store-and-forward and wormhole routing*, in In Proc. of the 28th ACM Symp. on Theory of Computing, 1996, pp. 356–365.

[25] U. Feige, *Nonmonotonic phenomena in packet routing*, in Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, New York, May 1–4 1999, ACM Press, pp. 583–591.

[26] U. Feige and P. Raghavan, *Exact analysis of hot-potato routing*, in Proceedings of the 33rd Annual Symposium on Foundations of Computer Science, IEEE, ed., Pittsburgh, PN, Oct. 1992, pp. 553–562.

[27] R. I. Greenberg and H.-C. Oh, *Universal wormhole routing*, IEEE Transactions on Parallel and Distributed Systems, 8 (1997), pp. 254–262.

[28] B. Hajek, *Bounds on evacutation time for deflection routing*, Distributed Computing, 1 (1991), pp. 1–6.

[29] C. Kaklamanis, D. Krizanc, and S. Rao, *Hot-potato routing on processor arrays*, in Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, Velen, Germany, June 30–July 2, 1993, pp. 273–282.

[30] M. Kaufmann, H. Lauer, and H. Schroder, *Fast deterministic hot-potato routing on meshes*, in Proceedings of the 5th International Symposium on Algorithms and Computation (ISAAC), LNCS, Springer-Verlag, ed., vol. 834, 1994, pp. 333–341.

[31] M. Kunde, *A new bound for pure greedy hot potato routing*, in Proceedings of the 24th Annual Symposium on Theoretical Aspects of Computer Science, vol. 4393 of Springer LNCS, Aachen, Germany, February 2007, pp. 49–60.

[32] F. T. Leighton, *Introduction to Parallel Algorithms and Architectires: Arrays-Trees-Hypercubes*, Morgan Kaufmann, 1992.

[33] F. T. Leighton, B. M., and A. W. Richa, *Fast algorithms for finding O(congestion + dilation) packet routing schedules*, Combinatorica, 19 (1999), pp. 375–401.

[34] F. T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao, *Randomized routing and*

sorting on fixed-connection networks, J. Algorithms, 17 (1994), pp. 157–205.

[35] F. T. Leighton, B. M. Maggs, and S. B. Rao, *Packet routing and job-scheduling in $O(congestion + dilation)$ steps*, Combinatorica, 14 (1994), pp. 167–186.

[36] F. Meyer auf der Heide and C. Scheideler, *Routing with bounded buffers and hot-potato routing in vertex-symmetric networks*, in Proceedings of the Third Annual European Symposium on Algorithms, Paul G. Spirakis, ed., vol. 979 of LNCS, Corfu, Greece, 25–27 Sept. 1995, pp. 341–354.

[37] F. Meyer auf der Heide and B. Vöcking, *Shortest-path routing in arbitrary networks*, Journal of Algorithms, 31 (1999), pp. 105–131.

[38] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[39] I. Newman and A. Schuster, *Hot-potato algorithms for permutation routing*, IEEE Transactions on Parallel and Distributed Systems, 6 (1995), pp. 1168–1176.

[40] R. Ostrovsky and Y. Rabani, *Universal $O(congestion{+}dilation{+}\log^{1+\varepsilon} N)$ local control packet switching algorithms*, in Proceedings of the 29th Annual ACM Symposium on the Theory of Computing, New York, May 1997, pp. 644–653.

[41] G. E. Pantziou, A. Roberts, and A. Symvonis, *Many-to-many routing on trees via matchings*, Theoretical Computer Science, 185 (1997), pp. 347–377.

[42] R. Prager, *An algorithm for routing in hypercube networks*, master's thesis, University of Toronto, Computer Science Department, 1986.

[43] Y. Rabani and É. Tardos, *Distributed packet switching in arbitrary networks*, in Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, 22–24 May 1996, pp. 366–375.

[44] P. Raghavan and C. D. Thompson, *Randomized rounding: A technique for provably good algorithms and algorithmic proofs*, Combinatorica, 7 (1987), pp. 365–374.

[45] R. Ramaswami and K. N. Sivarajan, *Optical Networks, a Practical Perspective*, Morgan Kaufmann, 1998.

[46] A. Roberts, A. Symvonis, and D. R. Wood, *Lower bounds for hot-potato permutation routing on trees*, in Proceedings of the 7th Int. Coll. Structural Information and Communication Complexity, SIROCCO, M. Flammini, E. Nardelli, G. Proietti, and P. Spirakis, eds., Carleton Scientific, 20–22 June 2000, pp. 281–295.

[47] C. Scheideler, *Universal Routing Strategies for Interconnection Networks*, vol. 1390 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, 1998.

[48] P. Spirakis and V. Triantafillou, *Pure greedy hot-potato routing in the 2-D mesh with random destinations*, Parallel Processing Letters, 7 (1997), pp. 249–258.

[49] A. Srinivasan and C-P. Teo, *A constant factor approximation algorithm for packet routing, and balancing local vs. global criteria*, in Proceedings of the ACM Symposium on the Theory of Computing (STOC), 1997, pp. 636–643.

[50] A. Symvonis, *Routing on trees*, Information Processing Letters, 57 (1996), pp. 215–223.

[51] L. Zhang, *Optimal bounds for matching routing on trees*, in Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, 1997, pp. 445–453.