Scheduling File Transfers in a Distributed Network

E. G. Coffman, Jr.

M. R. Garey

D. S. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

A. S. LaPaugh

Princeton University
Princeton, NJ 08544

ABSTRACT

We consider a problem of scheduling file transfers in a network so as to minimize overall finishing time, which we formalize as a problem of scheduling the edges of a weighted multigraph. Although the general problem is NP-complete, we identify polynomial time solvable special cases and derive good performance bounds for several natural approximation algorithms. The above results assume the existence of a central controller, but we also show how the approximation algorithms, along with their performance guarantees, can be adapted to a distributed regime.

1. INTRODUCTION

In this paper we study a fundamental problem of distributed processing, that of transferring large files between various nodes of a network. In particular, we are interested in how collections of such transfers can be scheduled so as to minimize the total time for the overall transfer process.

In our model, an instance of the problem consists of a labeled, undirected multigraph $G=(V,E)$, which we shall call the *file transfer graph*. Both the vertices and edges of this graph are labeled. Vertices correspond to computers/communication centers, each of which is assumed to have the ability to communicate directly with every other center. The label $p(v)$ of a vertex v is its *port constraint*, and denotes the maximum number of simultaneous file transfers that the given vertex can engage in. Edges correspond to the files to be transferred, with the label $L(e)$ of an edge e representing the amount of time needed to transfer that file. This is assumed to be independent of the time at which the file is transferred or the ports involved. Forwarding is not allowed; each file is transferred directly between the centers that are its endpoints. We also assume that once the transfer of a file begins, it continues without interruption until the transfer is complete.

Although this model was originally devised for one particular application, it is relevant to a variety of situations. For instance, consider a network of home computers, where each computer has an automatic dialer and interconnection is accomplished via a standard telephone network. This would correspond to our model with $p(v)=1$ for each vertex. On a larger scale, one could consider the computer centers of a large company, each having many automatic dialers.

The optimization criterion on which we concentrate, minimizing the *makespan* of the schedule (the time interval between the beginning of the first transfer and the completion of the last transfer), is relevant to all these situations. For instance, in the dial-up situations, one may wish to make all transfers during periods of low usage (or low telephone rates), and hence finding schedules which are "short" enough to fit into such intervals is a desirable goal.

In Sections 2 and 3 of this paper we shall consider the construction of schedules by a single *central controller* that, given the file transfer graph, constructs an overall schedule in advance. In Section 2 we present complexity results and algorithms for the problem of finding an optimal schedule under various restrictions on the problem instance. In Section 3 we analyze approximation algorithms that guarantee near-optimal solutions for those cases where finding an optimal solution is too difficult.

Results for a central controller that knows all the details of an instance give us an idea of the best we can hope for in any scheduling regimen. In Section 4 we consider what is possible in the perhaps more common case of distributed control, where the schedule is constructed on the fly in a distributed fashion, with each vertex knowing only its local part of the file transfer graph (and perhaps not all of that in advance), and with the possibility that vertices (and communication lines) may not be reliable. We conclude in Section 5 by listing some directions for future research.

2. COMPLEXITY RESULTS AND SOLVABLE SUBCASES

Given a file transfer graph $G=(V,E)$, a *schedule* can be viewed formally as a function $s: E \rightarrow [0,\infty)$ that assigns to each edge e a start time $s(e)$, such that, for each vertex v and time $t \geq 0$,

$$\left|\{e: v \text{ is an endpoint of } e \text{ and } s(e) \leq t < s(e)+L(e)\}\right| \leq p(v).$$

The *length* or *makespan* of a schedule s is the largest finishing time, i.e., the maximum, over all edges e, of $s(e)+L(e)$. Figure 1 illustrates file transfer graphs and the timing diagrams to be used in representing schedules. Our goal is to find, given a file transfer graph G, a schedule s with the minimum possible makespan.

An elementary lower bound is obtained as follows. Let E_u denote the set of files that are to be sent or received by vertex u. The degree of node u is given by $d_u = |E_u|$. Let $E_{u,v}$ denote $E_u \cap E_v$, the set of files to be communicated between u and v. We define $\Sigma_u = \sum_{e \in E_u} L(e)$ and $\Sigma_{u,v} = \Sigma_{e \in E_{u,v}} L(e)$. For consistency with this notation, we shall also use p_u to denote the port constraint $p(u)$ for each vertex u.

The time to transmit all of the files sent or received by vertex u is at least $\left\lceil \Sigma_u / p_u \right\rceil$. Thus we have the following:

Lemma 1. The optimal schedule length $OPT(G)$ for any graph G must satisfy

$$OPT(G) \geq \max_u \left\lceil \Sigma_u / p_u \right\rceil .$$

Note that this lower bound is achieved by the schedule in Fig. 1, where $T = \left\lceil \Sigma_{v_1}/p_{v_1} \right\rceil = 12/2 = 6$. Figure 2 illustrates the fact that $OPT(G)$ can be substantially larger than $\max_u \left\lceil \Sigma_u/p_u \right\rceil$. Here the value of the bound is 2, but the following simple analysis shows that a makespan of 2 is not achievable.

In order to finish the three files of any one triangle in 2 time units, a two-port vertex of that triangle has to transmit two of the triangle's three files simultaneously, during one or the other time unit. Since there are only 2 two-port vertices, two of the triangles must be transmitting two files in the same time unit. This can be done only by the first and third triangles, each transmitting the files incident with its 2-port vertex. But during this time unit no file of the middle triangle can be in transmission. Since this triangle requires 2 time units, we thus have $OPT(G) > 2$. In fact, it is easy to see that $OPT(G) = 3 = (3/2) \max_u \left\lceil \Sigma_u/p_u \right\rceil$.

It is not always "easy to see" what $OPT(G)$ is, however. The general decision problem "Given G and a bound B, is there a schedule s for G with makespan B or less?" is NP-complete and hence unlikely to be solvable efficiently, i.e., by a polynomial time algorithm. (For a discussion of NP-completeness and its implications, see [8].) Let us refer to this decision problem as FILE TRANSFER SCHEDULING. In the remainder of this section we will consider the complexity of this problem and some of its more interesting special cases.

Before we begin our search for polynomial time solvable subproblems of FILE TRANSFER SCHEDULING, it is important to note that there are two distinct reasons why the general problem is NP-complete, one having to do with the structural complexity of the network and the other with the complexity involved in balancing the transmission of files of different lengths. We shall illustrate this by proving that two very restricted versions of the problem are NP-complete.

Theorem 1. FILE TRANSFER SCHEDULING is NP-complete even when restricted to file transfer graphs in which the port capacity $p(v)$ of each vertex is 1, there is at most one edge between each pair of vertices, and all edges have the same length.

Proof. The problem is in NP because the general problem is: a nondeterministic algorithm need only guess a schedule and check (in polynomial time) that it is a valid schedule and meets the specified bound on makespan. Thus all we must show in this and future NP-completeness proofs is that some known NP-complete problem is polynomially transformable into the current one.

In this case the known NP-complete problem is CHROMATIC INDEX [9]: "Given a graph G and a positive integer k, can the edges of G be colored with k or fewer colors so that no two edges with the same color share a common endpoint?" This can be directly translated into the current special case, for if we view G as a file transfer graph with edges of length 1 and port capacities of 1, then an edge coloring corresponds to a schedule, with all the edges of the same color being scheduled during the same time unit and with the makespan equalling the total number of colors used. ■

Theorem 2. FILE TRANSFER SCHEDULING is NP-complete even when restricted to file transfer graphs with just two vertices, u and v.

Proof. In this case the transformation is from MULTIPROCESSOR SCHEDULING [8]: "Given a set T of *tasks*, each task t having an integer *length* $L(t)$, a number m of processors, and a bound B, can the tasks be partitioned among the m processors so that the total length of the tasks assigned to any one processor never exceeds B?"

In this case, we let $p(u) = p(v) = m$ and create $|T|$ edges between u and v, one for each task, letting the length of an edge equal the length of the corresponding task. It is then easy to see that a schedule with makespan B or less will exist if and only if the desired task partition exists. Note that, since MULTIPROCESSOR SCHEDULING is NP-complete for any fixed value of m greater than 1 [8], FILE TRANSFER SCHEDULING is thus NP-complete for any fixed bound of 2 or more on the port capacities. Like MULTIPROCESSOR SCHEDULING for an arbitrary number of processors, it is NP-complete "in the strong sense" [8] if port capacities are arbitrary. ■

Theorems 1 and 2 considered restrictions on FILE TRANSFER SCHEDULING involving edge lengths, the possibility of multiple edges between two vertices, bounds on port capacities, and the structure of the file transfer graph. We have conducted a more thorough investigation of the complexity of the problem in terms of these parameters, and our results are summarized in Tables I and II. Table I lists results for the case where all edge lengths are equal and hence structural considerations dominate; Table II considers the case where arbitrary edge lengths are allowed. The other parameters are as follows:

(1) File transfer graph structure; special cases include bipartite graphs, trees, cycles, and paths. Such restrictions might well arise in physical networks with restricted interconnection patterns, where nodes only communicate with their neighbors.

(2) Port constraints; a single port per vertex will be the special case, corresponding for example to the personal computer application mentioned above.

(3) Edge multiplicity; as a special case we will consider graphs having at most one edge between each pair of vertices. This restriction is often imposed in practice, e.g., when only a single telephone call between any pair of vertices is feasible. In this case an edge of the file transfer graph represents *all* the files to be transferred between its two endpoints.

EQUAL EDGE LENGTHS	One Port		Arbitrary Ports	
	Single Edges	Multiple Edges	Single Edges	Multiple Edges
General Graphs	NPC	NPC	NPC	NPC
Bipartite Graphs	$O(\|E\|\|V\|)$	$O(\|E\|\|V\|)$	$O(\|E\|^2)$	$O(\|E\|^2)$
Trees	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$
Paths and Even Cycles	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$
Odd Cycles	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$	?

Table I. Complexity classification for FILE TRANSFER SCHEDULING when all edge lengths are equal.

ARBITRARY EDGE LENGTHS	One Port		Arbitrary Ports	
	Single Edges	Multiple Edges	Single Edges	Multiple Edges
General Graphs	NPC	NPC	NPC	NPC
Bipartite Graphs	NPC	NPC	NPC	NPC
Trees	NPC	NPC	NPC	NPC
Paths and Even Cycles	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$	NPC
Odd Cycles	$O(\|E\|)$	NPC	$O(\|E\|)$	NPC

Table II. Complexity classification for FILE TRANSFER SCHEDULING when arbitrary edge lengths are allowed.

Entries in the tables are either "NPC" for NP-complete, "?" for "Open," or an upper bound on running time (in those cases where polynomial time algorithms have been found). The proofs of the results for paths and cycles will be left for the full paper, but note the

significant differences between the cases for even and odd cycles (even in those columns where the results are the same, more intricate proofs are needed for the odd case). We now show the derivation of the remaining results, which all follow from five basic results, the first two of which we have already given.

Theorem 1 implies the NP-completeness of all the special cases where general graphs are allowed, since all of these include the edge coloring problem as a special case. The polynomial time algorithms for the case of bipartite graphs when all edge lengths are equal are also based on the analogy with edge coloring, since that problem can be solved in polynomial time for bipartite graphs [1,3,7]. Known techniques will suffice for the multi-edge, single-port case, and a straightforward application of augmenting path arguments yields the following result when arbitrary port capacities are allowed.

Theorem 3 If G is bipartite and all edge lengths are equal, a minimum makespan schedule can be found in polynomial time, in particular in time $O(\min[|E|^2, |E|\cdot|V|\cdot\max_v p(v)])$.

The details of the proof are left for the full paper, and improvements in the running time are left to the reader (the above bound is obtained without using the more sophisticated techniques for handling augmenting paths discussed in [3,7], which seem likely to be applicable [6]).

This result immediately implies that all the remaining special cases in Table I, except those involving odd cycles, can be solved in polynomial time. However, those special cases can in fact be solved with even faster algorithms, as the following theorem shows for the case of trees.

Theorem 4. If G contains no simple cycles other than the trivial 2-vertex cycles induced by multiple edges (i.e., if G becomes a tree if multiple edges are coalesced), and if all edge lengths are equal, then a minimum makespan schedule can be found in time $O(|E|)$.

Proof. If G is not connected, the schedules for its connected components are independent and hence can be constructed separately. Thus all we need to show is how to do the scheduling if G is connected (and hence a tree). Without loss of generality assume that all edges have unit length. Recall that $\max_{u \in G} \left\lceil \Sigma_u / p_u \right\rceil$ is a lower bound on schedule length when G has arbitrary port constraints. Here, with unit-length edges, the bound specializes to $T = \max_{u \in G} \left\lceil d_u / p_u \right\rceil$. The algorithm below achieves T.

For each vertex assume that the edges (files) are ordered so that multiple edges to the same neighboring vertex are consecutive in the ordering. We label the edges of the tree as follows.

We begin with an arbitrary vertex, v, and label the edges in their given order with the first T non-negative integers, used cyclically in the sequence $0,1,\ldots,T-1,0,1,\ldots,T-1,0,1,\ldots$. Figure 3 shows an example.

Now suppose v is any vertex whose edges have all been labeled and w is a vertex adjacent to v, some of whose edges are not yet labelled. (Since G is a tree, we can proceed in such a way that the only edges involving w that are labelled are those with v as the other endpoint, i.e., those in $E_{v,w}$.) If j, $0 \leqslant j < T$, is the label of the last edge in $E_{v,w}$ to be labelled, then the labeling of w's remaining edges begins with $j+1$ (or 0 if $j = T-1$) and proceeds cyclically through the sequence $0,1,\ldots,T-1$ as before.

In this way we repeatedly choose a vertex w adjacent to a completely labeled vertex v and then complete the labeling for edges incident on w. It is easy to see that, no matter what order we choose for the vertices and edges consistent with the above description, the edges of each vertex u will be labeled in some sequence $i_1, i_2, \ldots, i_k$, where $0 \leqslant i_1 < T$ and $i_{j+1} = i_j + 1 \pmod T$. The number of times an integer appears in this sequence of d_u integers is at most $\left\lceil d_u/T \right\rceil$. By definition of T we have

$$\left\lceil d_u/T \right\rceil \leqslant \left\lceil \frac{d_u}{\max_v \left\lceil d_v/p_v \right\rceil} \right\rceil \leqslant \left\lceil \frac{d_u}{\left\lceil d_u/p_u \right\rceil} \right\rceil \leqslant \left\lceil \frac{d_u}{d_u/p_u} \right\rceil = p_u.$$

Thus, if we begin transmitting every file (edge) labeled i at time i, $0 \leqslant i < T$, we will have a valid schedule for G that is finished at time T and hence has T for its makespan. Since our algorithm obviously has a linear running time, the theorem is proved. ■

Theorems 3 and 4 conclude our discussion of Table I. As to Table II, the following theorem implies NP-completeness for all the cases where the file transfer graph is at least a tree, and hence shows that the efficient algorithms of Theorems 3 and 4 for bipartite graphs and trees do not carry over to the case where arbitrary edge lengths are allowed.

Theorem 5. FILE TRANSFER SCHEDULING is NP-complete even when G is a tree of single port vertices without multi-edges.

Proof. Our reduction uses the 3-PARTITION problem: Given $A = \{a_1,\ldots,a_{3k}\}$, with $\frac{B}{4} < a_i < \frac{B}{2}$, $1 \leqslant i \leqslant 3m$, where $B = \frac{1}{k}\Sigma_{i=1}^{3k} a_i$, does there exist a partition $A = A_1 \cup A_2 \cup \ldots \cup A_k$ such that $\sum_{a \in A_i} a = B$, $1 \leqslant i \leqslant k$? (Note that this is again a special case of MULTIPROCESSOR SCHEDULING.) Let $C_i = iB + i - 1$, $1 \leqslant i \leqslant n$, and assume without loss of generality that k is odd and $k = 2n+1$. Given an instance of the above form, we construct a file transfer graph as illustrated in Fig. 4.

The tree is constructed around a hub vertex v, on which are incident (i) $3k$ edges with lengths a_1 through a_{3k} and (ii) n subtrees G_1 through G_n, each G_i having two length-1 edges, b_i and b_i', incident with v. Let $M = \Sigma_v = \Sigma_{i=1}^{3k} a_i + 2n$. (Note that M is a lower bound on the optimum makespan, since v has a single port.) The lengths of the remaining edges in G_i are as follows (for simplicity we shall use the same symbol to denote both the edge and its length):

$$g_i = g_i' = M - C_i, \quad 1 \leqslant i \leqslant n$$

$$f_i = f_i' = C_i, \quad 1 \leqslant i \leqslant n$$

$$e_i = e_i' = C_i + 1, \quad 1 \leqslant i \leqslant n$$

$$d_i = d_i' = M - C_i - 1, \quad 1 \leqslant i \leqslant n$$

We shall show that this tree of single-port vertices can be scheduled with makespan equal to the lower bound M if and only if the desired 3-partition exists. NP-completeness will then follow in the usual way.

Consider the scheduling of G_i, for any i, under the assumption that G can be scheduled in time M. Clearly, since $d_i + e_i = M$, transmission of d_i must either begin at $t = 0$ or end at $t = M$. By similar arguments, this must also hold for f_i, d_i', and f_i'. Since d_i and f_i cannot both start at time 0 because they share an endpoint, one must start at time 0 and the other end at time M. This means that b_i must be scheduled in the interval between them, either at time C_i or at time $M - C_i - 1$, depending on whether f_i or d_i goes first. A similar argument holds for b_i', so between the two tasks, each timeslot $[C_i, C_i + 1]$ and $[M - C_i - 1, M - C_i]$ must be occupied by one of the edges b_i and b_i', as in Fig. 5a.

Applying this argument for all i, $1 \leqslant i \leqslant n$, we deduce that the schedule at vertex v must look as in Fig. 5b, when restricted to tasks of types b and b'. Noting that $C_{i+1} - (C_i + 1) = B$, $1 \leqslant i < n$, and that $[M - (nB + n)] - (nB + n) = M - 2nB - 2n = kB + 2n - 2nB - 2n = B$, we conclude that there are $2n + 1 = k$ gaps left in the schedule, each of length exactly B, and into which the tasks of type a_i must be scheduled. It thus follows that G can be scheduled with makespan M if and only if there is a 3-PARTITION of A. ■

We note in concluding this section that the proofs of Theorems 1 and 5 actually prove NP-completeness "in the strong sense" with all that implies [8], although those of Theorems 2 and 8 do not. This leaves open the possibility of "pseudo-polynomial time" algorithms [8] in the latter two cases. In fact it is easy to see that one exists in the case of two vertices (Theorem 2), although it is not clear how far such an approach can be generalized.

3. APPROXIMATION ALGORITHMS

In the previous section we identified a number of special cases where minimum makespan schedules can be found efficiently. In general, however, NP-completeness blocks our way to finding optimal schedules efficiently. A standard approach to use when confronted with such an obstacle is to search for good "approximation algorithms," algorithms that efficiently generate schedules, but that provide no guarantee of optimality (although a guarantee of "near-optimality" may be possible). In this section we shall consider such algorithms, showing how classical results from the study of multiprocessor scheduling can be adapted to the more general problem of file transfer scheduling, and can provide both efficient algorithms and reasonable guarantees.

The main algorithm we shall study is *List Scheduling (LS)*. This algorithm assumes that the edges of the file transfer graph are ordered in a list as $e_1,e_2,...,e_m$. At time $t = 0$, and thereafter at each time t an edge is finished, the list is scanned for "ready" edges, i.e. edges that have not yet been started and for which there is an available port at both endpoints. Whenever such an edge is encountered on the list, it is assigned starting time t, the two ports it uses become unavailable, and the scanning of the list is continued from that point. The list is always scanned in the given order, and ready edges are never delayed. The obvious implementation for List Scheduling is $O(|E|^2)$, since the list might have to be scanned once for each edge scheduled. However, cleverer implementations can reduce this bound to $O(|V||E|)$. (The basic trick is to spend $O(|V|)$ time updating the set of ready edges each time an edge finishes and another $O(|V|)$ in deciding which ready edge has priority each time an edge starts, followed by $O(|V|)$ to demote edges that are no longer ready.)

As illustrated in Fig. 6 schedule lengths resulting from different lists can differ substantially. Note that the example is easily generalized to one for which $d_0 = k$, $d_i = k-1$, $1 \leq i \leq k$, and the schedule lengths corresponding to lists $(e_1,...,e_k,e_{k+1},...,e_{k^2})$ and $(e_{k+1},...,e_{k^2},e_1,...,e_k)$ are k and $2k-1$, respectively. This file transfer graph will be given the special designation H_k; it will come up again later in this section.

Given a file transfer graph G and a sequencing S of the edges, denote by $LS(G,S)$ the makespan of the schedule produced when LS is applied to G with list S. It is easily verified that the complexity results of the last section also apply to the problem of finding an ordering of the edges of a file transfer graph that minimizes $LS(G,S)$. However, as the next result shows, even the best list schedule need not always be optimal. Define $OPT_{LS}(G) = \min_S LS(G,S)$, and let $OPT(G)$ be the optimal makespan for G.

Theorem 6. For any $\delta > 0$ there exists a file transfer graph G such that

$$OPT_{LS}(G) > \left(\frac{4}{3} - \delta\right) OPT(G)$$

Proof. Consider the graph in Fig. 7, where edges a_1, a_2, and a_3 have length $1+\epsilon$, $\epsilon << 1$, and all other edges have length 1. Each of the three subtrees incident on vertex u can be scheduled with makespan $3+3\epsilon$ by scheduling in parallel b_{i1} with e_{i1}, b_{i2} with c_{i2}, and a_i with c_{i1} and e_{i2} in the intervals $[0,1+\epsilon]$, $[1+\epsilon,2+2\epsilon]$, and $[2+2\epsilon,3+3\epsilon]$, taken in any order. Adopting an order for the ith subtree that schedules a_i, c_{i1} and e_{i2} in interval $[(i-1)+(i-1)\epsilon, i+i\epsilon]$, $i = 1,2,3$, therefore assures that the subtrees incident on u can be scheduled in parallel, i.e. a port conflict at u can be avoided. Thus, the graph can be scheduled in time $3+3\epsilon$. Note, however, that the schedules above are not list schedules; delays (of at most ϵ) have been introduced at times when there are ready edges.

Let us now suppose that there is an ordering S for G such that $LS(G,S) < 4$. Since all edges other than the a_i's have unit lengths, the earliest of the a_i's to start, say a_1, must start at an integer time. If a_1 started at a time $t \geq 1$, then since the a_i's must be scheduled in disjoint intervals, the length of the schedule would be at least $4+3\epsilon$. Thus, a_1 must start at $t = 0$ along with one of c_{11} and c_{12} and one of e_{11} and e_{12}, say c_{11} and e_{11}. At $t = 1$, a_1 is not finished so b_{11} and b_{12} cannot start. Thus, c_{12} and e_{12} must start, and this prevents b_{11} and b_{12} from starting prior to $t = 2$. Since b_{11} and b_{12} must be scheduled in disjoint intervals we have immediately $LS(G,S) \geq 4$. Thus,

$$\frac{OPT_{LS}(G)}{OPT(G)} \geq \frac{4}{3+3\epsilon}$$

and the theorem follows by choosing ϵ sufficiently small. ■

Fortunately, although the best list schedule may not be all that close to optimal, the worst one cannot be all that far away.

Theorem 7. For any file transfer graph G and any list S of its edges, we have

$$LS(G,S) \leq 2OPT(G) + \max\left\{0, L\left[1-\frac{2}{p}\right]\right\},$$

where L is the maximum edge length and p is the maximum port capacity. Moreover, the bound is tight in the sense that for any value of p and for values of $L/OPT(G)$ as large as 1 (if $p \geq 2$) or values of $L/OPT(G)$ approaching 0 (if $p = 1$), there are graphs (in fact trees) such that the upper bound is attained for some ordering S.

Proof. In any given list schedule of G, let $e = \{u,v\}$ denote an edge with latest finishing time, and consider the vicinity of e in G as shown in Fig. 8. At any time prior to the start of e, it must be the case that either all ports at u or all ports at v are busy. But all ports at u can be busy, with edges other than e, for at most $\lfloor(\Sigma_u - L(e))/p_u\rfloor$ units of time, and the symmetric bound holds for v (recall that all edge lengths are integers). Thus, since $LS(G,S)$ equals the starting time for e plus its length $L(e)$, we have

$$LS(G,S) \leq \left\lfloor\frac{\Sigma_u - L(e)}{p_u}\right\rfloor + \left\lfloor\frac{\Sigma_v - L(e)}{p_v}\right\rfloor + L(e) \qquad (**)$$

$$\leq \frac{\Sigma_u}{p_u} + \frac{\Sigma_v}{p_v} + L(e)\left(1 - \frac{1}{p_u} - \frac{1}{p_v}\right)$$

Since $OPT(G)$ cannot be less than either Σ_u/p_u or Σ_v/p_v, L cannot be less than $L(e)$, and p_v and p_u cannot exceed p, the upper bound follows.

If $p = 1$, then $1 - 2/p = -1$, and the upper bound becomes $2OPT(G) - L$. This can be obtained via the trees H_k illustrated in Figure 6 for $k = 4$. As described in the text, the maximum edge length in H_k is $L_k = 1 = OPT(H_k)/k$ and there are orderings such that $LS(H_k,S) = (2-1/k)OPT(H_k) = 2OPT(H_k) - L_k$. Letting k go to ∞, we see that this bound can be obtained for arbitrarily small values of $L/OPT(G)$.

Let us now turn to the case where $p \geq 2$. For any such p, we can construct a tree and an ordering S of its edges such that $LS(G,S)$ equals the quoted bound. See Figure 9. This graph is built around a single edge $e = \{u,v\}$ of length L, where L will turn out to equal $OPT(G)$. To the endpoint u of e, which has port constraint p, are attached $p(p-1)$ length-1 edges $a(i)$, while to the other endpoint v, also with port constraint p, are attached $p-1$ trees $G(i)$, each $G(i)$ being a copy of the tree $H(p)$, all of whose internal vertices have port constraint 1 and all of whose edges have length 1. It is easy to see that we can order the edges so that a schedule of the following form is produced by LS.

In each of the first $p-1$ time units we schedule one edge from each of the sets $c_{jk}(i)$ $(1 \leq j \leq p, 1 \leq k \leq p-1)$ in each of the $G(i)$. Thus a total of $p(p-1)$ of these edges are scheduled in each time unit of $[0,p-1]$. Also starting at $t = 0$, all of the edges a_i, $1 \leq i \leq p(p-1)$, are scheduled. Since the port constraint at vertex u is p, the a_i's are scheduled p at a time and thus occupy the first $p-1$ time units. Thus, edge e cannot be scheduled during $[0,p-1]$, and at $t = p-1$ we are left with the tree consisting of e and the $p(p-1)$ edges $b_j(i)$ $(1 \leq i \leq p-1, 1 \leq j \leq p)$.

Starting at $t = p-1$ we schedule all of the $b_j(i)$'s; since the port constraint at vertex u is p they are scheduled p at a time in the interval $[p-1,2(p-1)]$. The start of edge e is clearly delayed until

$t = 2(p-1)$. Since $L(e) = p$, we thus have $LS(G,S) = 3p-2$.

An optimal schedule for the graph is easily found. Each of the trees $G(i)$, $1 \leq i \leq p-1$, is scheduled optimally as shown in Figure 6 for H_k. Since the port constraint at vertex u is p, edge e and each of the $p-1$ $G(i)$'s can be done in parallel in the first p time units. During $[0,p]$ the $p-1$ ports left available by e at vertex v are used to schedule the $p(p-1)$ a_i's, $p-1$ at a time. This schedule is obviously a best possible schedule, and hence $OPT(G) = p = L$. We thus have

$$LS(G,S) = \left\lceil \frac{3p-2}{p} \right\rceil OPT(G) = 2OPT(G) + L\left(1 - \frac{2}{p}\right)$$

with $L/OPT(G) = 1$, as desired. ■

The significance of Theorem 7 is best seen in terms of the following corollaries.

Corollary 7.1. For any file transfer graph G and all orderings S of the edges of G,

$$\frac{LS(G,S)}{OPT(G)} < 3,$$

and this bound is tight in the sense that there are trees G and orderings S such that $LS(G,S)/OPT(G)$ is arbitrarily close to 3.

Proof. Since $OPT(G)$ is always at least as large as the maximum edge length L of G, the upper bound follows from the upper bound of Theorem 7. The lower bound examples are those of Theorem 7 for arbitrarily large values of p. ■

Corollary 7.2. If the maximum port capacity p for G is 2 or less, then

$$\frac{LS(G,S)}{OPT(G)} \leq 2$$

and this bound is tight in the sense that there are trees G obeying these port capacity bounds and orderings for which the ratio is arbitrarily close to 2.

Proof. If $p \leq 2$, then $1-2/p \leq 0$, so the upper bound once again follows from Theorem 7. The lower bound examples are the trees H_k, as k goes to ∞. ■

Corollary 7.3. If all edges in G have the same length, then

$$\frac{LS(G,S)}{OPT(G)} < 2$$

and this bound is tight in the sense that there are trees of equal length edges and orderings for which the ratio is arbitrarily close to 2.

Proof. We may assume without loss of generality that all edge lengths are 1, in which case $\Sigma_v = d_v$, and the argument behind (**) leads to

$$\begin{aligned} LS(G,S) &\leq \left\lceil \frac{d_u - 1}{p_u} \right\rceil + \left\lceil \frac{d_v - 1}{p_v} \right\rceil + 1 \\ &< \left\lceil \frac{d_u}{p_u} \right\rceil - 1 + \left\lceil \frac{d_v}{p_v} \right\rceil - 1 + 1 \\ &< \left\lceil \frac{d_u}{p_u} \right\rceil + \left\lceil \frac{d_v}{p_v} \right\rceil \leq 2 \cdot OPT(G) \end{aligned}$$

The lower bound examples are once again the trees H_k as k goes to ∞. ■

List Scheduling algorithms, such as the one we have been discussing, have the drawback that they are at the mercy of whoever or whatever is ordering the edges. In the above proofs we used this fact in constructing our worst-case examples. The question naturally arises, what advantage might we gain if we had the power to order the edges ourselves? We saw in Theorem 6 that it may be that no re-ordering of the list of edges will yield an optimal schedule. However, a very natural ordering, easy to compute and often used to good advantage in other scheduling problems, does yield an improved bound. The idea is to re-order the edges in "decreasing" order, i.e., so that $L(e_1) \geq L(e_2) \geq \cdots \geq L(e_m)$. Because of ties between edges of equal length, there may be more than one possible decreasing order, so we shall evaluate this approach with a conservative, "worst-case" measure. Let

$DLS(G) \equiv \max\{LS(G,S): S \text{ is a decreasing ordering of the edges of } G\}$.

We then have the following result.

Theorem 8. For any file transfer graph G with maximum port capacity $p \geq 2$,

$$\frac{DLS(G)}{OPT(G)} \leq \left\lceil \frac{5}{2} - \frac{1}{p} \right\rceil$$

Moreover, this bound is tight in the sense that for each p there exist trees and orderings for which the ratio approaches the given bound.

Proof. Consider Fig. 8 in Theorem 7, where e is the last edge to finish in our decreasing-list schedule. If $DLS(G) > OPT(G)$, e cannot have started at time 0. Therefore, at time 0 either all the ports of vertex u were busy or else all the ports of vertex v were busy. Assume without loss of generality that the latter is true. Since this is a decreasing-list schedule, this means that v must be an endpoint for at least p_v edges $e' \neq e$ with $L(e') \geq L(e)$. But this means that v is an endpoint of $p_v + 1$ edges of length at least $L(e)$, and hence at least two of them must be scheduled in disjoint intervals, which means that $OPT(G) \geq 2L(e)$. Substituting this inequality into Theorem 7 gives the desired upper bound.

We leave the proof that the bound is tight for the full paper. The proof is considerably more difficult than that for the lower bound in Theorem 7 (indeed, we initially suspected that an upper bound of 2 might hold here, rather than the 2.5 of the theorem). The construction involves trees for which $|V|$ is a rapidly growing exponential function of p, and takes a comparable number of pages to describe. ■

We have been unable to find any general polynomial time approximation algorithm that provides an improvement on the $2.5OPT(G)$ bound for DLS. However, we have been able to provide better guarantees in special cases. For instance, if all edge lengths are equal and their are no multi-edges, one can apply Vizing's result [1,12] that any graph can be edge-colored using only one more color than the obvious lower bound. The proof of this result yields a polynomial time algorithm for generating such a coloring, which in turn yields a scheduling algorithm that is guaranteed to come within an *additive* constant of $OPT(G)$ in the case where all port capacities are 1. If multi-edges are allowed, we can still improve on DLS in the single port case, although the additive constant now is multiplied by the maximum edge multiplicity. For high edge multiplicities, this can be replaced by an algorithm guaranteeing $\left\lceil \frac{3}{2} OPT(G) \right\rceil$ that is based on an edge-coloring result of Shannon [1,11]. (Recall that DLS only guarantees $2OPT(G)$ for such instances.)

In the more likely situation of arbitrary edge lengths, we can still improve on DLS so long as the file transfer graph is a tree, where recall that DLS can still be as bad as $2.5OPT(G)$ (even if all port capacities equal 1). In the full paper we show how to adapt the MULTIFIT DECREASING algorithm for multiprocessor scheduling [2,5] to provide guarantees of $2.2OPT(G)$ if G is a tree without multi-edges and of $2.4OPT(G)$ if G is a tree with multi-edges but with all port capacities equal. These two results are applications of general methods we have developed for adapting multiprocessor scheduling algorithms to these types of trees. If instead of MULTIFIT DECREASING we use the fixed processor scheduling algorithms of [10], we can obtain polynomial time guarantees of $(2+\epsilon)OPT(G)$ for any fixed ϵ in both cases, so long as the maximum port capacity obeys a fixed bound (the degree of the polynomial depends on the bound).

4. DISTRIBUTED SCHEDULING

The scheduling algorithms of the previous section have all assumed the existence of some central processor that knows in advance all the files to be transferred and their lengths. An even more basic assumption is that once a schedule has been constructed it can be carried out without a hitch: the various nodes will be able to commence sending and receiving files at the predetermined times and all transfers will take the predicted time. In many of the envisioned applications of

our problem, neither of these assumptions will apply. No central processor will exist, transmittal times may not be known exactly in advance, and indeed the set of files to be transmitted may not be entirely known when the first transmittal takes place. The question thus arises: How much do our results for the idealized file transfer scheduling problem have to say about the actual transfer of files in distributed systems?

The NP-hardness results continue to tell us something. If finding optimal makespan schedules is difficult when one processor knows everything in advance, it certainly does not become any easier when the problem must be solved in a distributed fashion while subject to imperfect information. Surprisingly, however, there is also some carry-over of our algorithmic results, in particular, those concerning the approximation algorithm List Scheduling. The algorithm itself, requiring a central processor, does not apply. However, the proof of its guarantee can be adapted to yield a similar result for a kind of schedule that *can* be generated by a distributed process. (Note that any process for transferring files, even one that does not involve preassigned starting times, can be viewed as generating a schedule *ex post facto*; one need only observe when transfers actually began and how long they took to define both the schedule and the file transfer graph.)

Let us call a schedule s for file transfer graph G a *demand schedule* if there is no interval of time $[t,t']$, $0 \leq t < t'$, such that for some vertices u and v with an edge e between them, e does not start before t' even though both u and v have an idle port during the interval $[t,t']$. If s is not a demand schedule, define for each pair u,v of distinct vertices in G the quantity $delay_s(u,v)$ to be the total length of intervals during which the above is violated for u and v. A schedule s will be called a *Δ-delayed demand schedule*, $\Delta \geq 0$, if Δ equals the maximum value of $delay_s(u,v)$ for all pairs u,v. Note that a 0-delayed demand schedule is simply a demand schedule. A careful examination of the proof of Theorem 7 shows that it implies the following:

Theorem 9. If s is a Δ-delayed demand schedule for G, then

$$MAKESPAN(s) \leq 2OPT(G) + L(e)\left(1 - \frac{1}{2p}\right) + \Delta$$

$$\leq 3OPT(G) + \Delta$$

where e is the last edge to finish and p is the maximum port capacity.

Now every schedule is a Δ-delayed demand schedule for some Δ; what we need are schedules with small Δ's. Fortunately, distributed scheduling algorithms are fairly easy to devise that construct Δ-delayed demand schedules for Δ's that one can reasonably expect to be small in relation to the overall makespan. Consider the application in which ports correspond to telephone lines, as in a network of home computers. The only way to communicate with another node is to call it up, using the same phone lines over which the files themselves are to be transferred. If a vertex has more than one port, we shall assume that an automatic call routing mechanism sends an incoming call to a free port if one exists. The following distributed algorithm suggests itself. It assumes that file transfers, once initiated, operate in parallel with the scheduling protocols, terminating when the transfer is finished by disconnecting the phone link and freeing the port used for the transfer.

Each vertex v has a set E_v of the edges (files) it wants to send or receive. (We do not require that both endpoints of an edge have that edge in their sets, only that one of them does.) Each vertex v maintains a queue Q_v of those files it currently wants to send or receive. Each vertex v then executes the following protocol:

DEMAND PROTOCOL 1 (DP1).

Repeat until Q_v is empty:

1. If v has an idle port, attempt to call the other endpoint u of the first edge e in Q_v.
   - 1.1. If u answers, initiate the transfer of e and delete e from Q_v.
   - 1.2. If busy or no answer, move e to the end of the queue.
2. If v has an idle port, wait some prespecified amount of time ϵ for an incoming call.
   - 2.1. If a call is received from a neighbor u, initiate the requested transfer, delete the corresponding edge e from Q_v (if it was present), and end Step 2.
   - 2.2. If no call is received after waiting for ϵ, end Step 2.

END (DEMAND PROTOCOL 1)

The main problem faced here does not concern file transfer scheduling but rather the difficulties of communication inherent in this distributed "dial-up" model. How does one locate a neighbor with a free port when there is always the possibility of "false" busy signals, i.e., busy signals caused because the node being called is itself making a call that will eventually receive a busy signal or no answer? By appropriately adjusting the length of the wait in Step 2 (with perhaps different values for different vertices), one can presumably minimize the amount of time one expects to spend in such churning. (Step 1.2, by moving an edge that got a busy signal to the end of the queue, prevents us from endlessly calling a truly busy neighbor when idle neighbors still exist.) More complicated schemes for determining when one calls and when one waits might yield further improvements. However, since the avoidance of busy signals is not the main subject of the current paper, we shall merely attempt to isolate this "communication delay" from the delays caused by the scheduling heuristic.

Given a protocol P and a distributed network represented by a file transfer graph G, let $\delta(P,G)$ be the maximum time for the network, using P at each node, to initiate some transfer, given that there is an untransmitted edge whose endpoints both have free ports. Although theoretically no such upper bound may exist that holds over *all* executions of the protocol, such a bound can at least be computed *ex post facto* for any particular scheduling of G using P (assuming there is no infinite sequence of false busy signals).

The situations of interest are, of course, those in which $\delta(P,G)$ is small relative to $OPT(G)$. In this case the following corollary of Theorem 9, which follows from that result and the above definitions, becomes relevant.

Corollary 9.1. If s is a schedule constructed by Demand Protocol 1 with *ex post facto* file transfer graph G, maximum port capacity p, and last-finishing edge e, then

$$MAKESPAN(s) \leq 2OPT(G) + L(e)\left(1 - \frac{1}{2p}\right)$$

$$+ \delta(DP1,G)\cdot|E|$$

$$\leq 3OPT(G) + \delta(DP1,G)\cdot|E|\,.$$

Thus if communication delay can be kept small relative to file transfer time, the guarantees of the previous section can be carried over to the distributed milieu. Moreover, given any specified scheme for avoiding unnecessary busy signals, there is a sense in which DP1 can be expected to minimize communication delay, since it requires only one completed call per pair involved in a file transfer. The number of completed calls required could be further reduced by transferring *all* files that are to go between u and v as soon as the two vertices are first connected. However, this latter approach may have its own drawbacks. It would prevent the parallel transfer of multiple files between the same pair of vertices, and hence could give rise to a Δ-delayed demand schedule where Δ exceeds $\delta(P,G)\cdot|E|$, at least in those cases where port capacities exceed 1.

Note that protocol DP1 does not require that the lengths of the files be precisely known in advance. The protocol can also be easily adapted to the case where the required file transfers are not all known at the start of the transfer procedure, but arrive on-line, thus being added to the lists Q_v in midstream and causing the protocol to be restarted if Q_v was already empty at the time. (An explicit mechanism for this is illustrated in the next protocol we present.) The value of the Theorem 9 guarantee is considerably lessened, however, since now

delays can occur because of the late arrival of a transfer request, not just because of communication delays. However, one can still get a guarantee:

Corollary 9.2. If s is a schedule constructed by Demand Protocol 1 (modified to handle late arrivals), G is the *ex post facto* file transfer graph, and if one takes $OPT(G)$ to be the minimum makespan possible, given the late arrival of certain requests, then

$$MAKESPAN(s) < 4OPT(G) + \delta(DP1,G)\cdot|E|.$$

Proof. Simply partition the schedule into the part *before* the arrival of the last transfer request, which clearly has length less than $OPT(G)$, and the part starting when the last file transfer request arrives, from which point Theorem 9 applies with Δ equal to the remaining communication delay. ■

The protocol can also be adapted to tolerate unreliable vertices in a manner similar to that in the next protocol, although there is some question as to how appropriate makespan is as an optimization criterion in either the on-line situation or the case of unreliable processes – see Section 5. (As it stands, DP1 can handle a vertex that dies and stays dead, in which case it still insures that all transfers not involving the moribund vertices are eventually completed, assuming that there is no infinite sequence of false busy signals and that the death of a neighbor frees all ports it was using at the time).

If the system is such that communication delays are very small in comparison to file transfer time, one might be willing to spend more time in protocol communication in exchange for an improved bound such as the $(2.5)OPT(G)$ of Decreasing List Scheduling. In the remainder of this section we describe a protocol that does just that. As with the previous protocol, this one does not depend on precise knowledge of the edge lengths in the file transfer graph. (However, the 2.5 result will require that both endpoints of an edge know of its existence at time 0 and have identical estimates of its length. The extent to which an estimate can be "off" will determine the quality of the guarantee.) The protocol we describe can also handle on-line requests and tolerate unreliable vertices, although these may degrade its performance and the guarantees we prove will depend on the system behaving sensibly and reliably.

To focus on scheduling issues rather than the problems inherent in distributed communication, we shall push issues of communication and reliability even further into the background than we did for DP1, by postulating the existence of certain lower level protocols that behave as follows:

(1) There is a *Communications Protocol* that any vertex can invoke to send a query to a neighbor, and that will return an appropriate response (one of those specified by our scheduling protocol) or a "busy" signal, the latter only occurring if the neighbor is currently using all its ports for file transfers (or is either dead or so slow in responding that it *appears* to be dead).

(2) There are file transfer protocols *FileSend* and *FileReceive* that, given an agreement between two vertices to transfer a file, can be invoked by the sender and receiver respectively to effect the transfer. (The order of invocation is irrelevant; they take care of their own synchronization.) On completion of the transfer (or premature termination), each protocol informs its invoker of the transfer's success or failure.

We also assume that queries, responses, and reports on successful or unsuccessful file transfers are all placed on a First-In, First-Out *Protocol Message Queue* as they arrive, along with any new internally-generated requests for transfers.

Our old protocol could be reinterpreted in this model, with the one type of query being "Will you agree to perform a transfer?" and the allowed answers being "No" (when all my ports are full or I myself have an outstanding request) or "Yes" (in which case the requester and the answerer both initiate the appropriate file transfer protocols). However, note that such an implementation wastes the power of our assumption that all queries receive answers, by making one of the answers equivalent to receiving a busy signal in our original model. The power of that assumption, as we shall see, is that it allows us to prove that there cannot be an infinite sequence of operations without progress being made, as is at least possible in the original model.

The protocol we now describe is the distributed analogue of List Scheduling. Based on our results about it we can derive a result analogous to that for Decreasing List Scheduling as a corollary. Corresponding to the "list" in List Scheduling, there will be a total ordering "$>_G$" on the edges (including late arrivals), with the "greatest" edge corresponding to the first edge in the list. We assume that an individual vertex can determine the relative ordering between any two edges of which it is an endpoint.

The protocol uses just one type of query: "Are you prepared to perform the transfer represented by edge e?," which we shall denote by *Query* (v,e,u), where e is the name of the edge and u and v are its endpoints, v being the vertex sending the query, and u being the addressee. There are three permitted answers: (1) "Yes," denoted *Yes* (e,u), (2) "Call back later, I'm waiting to see whether I can execute a higher priority edge," denoted *Later* (e,u), and (3) "No, all my ports are in use (or I am dead)," denoted *No* (e,u). The invocation of the FileSend and FileReceive protocols for edge e are denoted by *FileSend* (e) and *FileReceive* (e) respectively. Reports from these protocols are of the form *Success* (e) or *Failure* (e). An internal request for a new transfer represented by the edge e has the form *Add* (e).

Each vertex v maintains two lists of incident edges. The first is the *A–list* and contains those edges whose transferral has not yet begun and whose other endpoints are not thought to be busy. It is ordered according to $>_G$ with the largest element first. The second is the *B–list* and contains all the rest of the as yet untransferred edges, ordered in a First-in, First-out fashion. We assume that these are initialized so that the B-list is empty and all transfer requests initially known by (and involving) v are in the A-list, marked as "unqueried," i.e., as not being the subject of any query sent by v but not yet answered. In addition, the variable $N_{QUERIED}$, initialized to 0, gives the current number of outstanding unanswered queries, and the variable $N_{FREEPORT}$, initialized to p_v, contains the current number of free ports. We shall assume $N_{QUERIED}$ is updated automatically, whereas we shall explicitly update $N_{FREEPORT}$ so that it reflects the number of ports *committed* to transfers, even though some of the transfers may not yet be taking place.

We are now in a position to describe the protocol.

DEMAND PROTOCOL 2 (DP2).

Repeat forever:

1. If the Protocol Message Queue is not empty, remove the first message M and do the following:
   - 1.1. If M = *No* (e,u), move e from its current position to the end of the B-list and mark it as "unqueried."
   - 1.2. If M = *Later* (e,u), mark e as "unqueried" and move it to the A-list if it is not already there.
   - 1.3. If M = *Yes* (e,u) then
     - 1.3.1. If e is in either the A-list or B-list, delete e from its list, reduce $N_{FREEPORT}$ by 1, and invoke the appropriate one of *FileSend* (e) and *FileReceive* (e).
     - 1.3.2. If e is not in either list, do nothing. (We have already invoked the appropriate one of *FileSend* (e) and *FileReceive* (e) in response to a query from u.)
   - 1.4. If M = *Query* (u,e,v) then
     - 1.4.1. If $N_{FREEPORT} = 0$, send the message *No* (e,v), add e to the A-list marked "unqueried" (if it is not already there), and delete it from the B-list (if present).
     - 1.4.2. Else if e is in the A- or B-list marked "queried," or if $N_{FREEPORT}$ exceeds the number of edges e' that either (i) are currently marked "queried" or (ii) are marked "unqueried," satisfy $e' >_G e$, and are in the A-list, then send the message *Yes* (e,v) to u, delete e

from its list (if it is on one), reduce $N_{FREEPORT}$ by 1, and invoke the appropriate one of *FileSend* (e) and *FileReceive* (e).

1.4.3. Else send the message *Later* (e,v) to u, add e to the A-list marked "unqueried" (if it is not already there), and delete it from the B-list (if present).

1.5. If M = *Success* (e), increase $N_{FREEPORT}$ by 1.

1.6. If M = *Failure* (e) or M = *Add* (e), add e to the A-list marked "unqueried" (if not already there), delete it from the B-list (if present), and increase $N_{FREEPORT}$ by 1.

2. While $N_{FREEPORT} > N_{QUERIED}$ and there is an edge in the A-list or the B-list that is marked "unqueried," do the following

2.1. Let e be the first edge marked "unqueried" in the concatenation of the A-list followed by the B-list, and let u be its other endpoint.

2.2. Send *Query* (v,e,u) and mark e in its list as "queried".

END (While $N_{FREEPORT}$ ...)

END (DEMAND PROTOCOL 2)

It is easy to verify (by induction or case analysis) that certain elementary properties are obeyed by this protocol. For instance, $N_{QUERIED}$ is never more than $N_{FREEPORT}$; a vertex, once informed of the existence of an edge, never forgets it until the corresponding transfer is successfully completed (if we count the fact that the file transfer protocols "remember" the edge while a transfer is being attempted, and reinstate it should the transfer fail); if one endpoint of an edge invokes a file transfer protocol for an edge, then so must the other unless it dies, i.e., stops making steps in its protocol loop. Our major claim, which follows from such elementary observations, is the following "correctness" result.

Theorem 10. Suppose that the network is being governed by DP2 and is in a state where there is an unstarted edge, both of whose endpoints are alive and have a free port. Then so long as no vertex dies and the Communications and File Transfer protocols operate as assumed, one of the following must happen: (a) an edge starts, (b) an edge finishes, or (c) a new edge gets internally requested at some vertex.

Proof. Suppose we are in a configuration where none of (a), (b), or (c) will ever occur again and no further vertices will die, and that, contrary to the theorem, there is an unstarted edge, both of whose endpoints are alive and have free ports (in what follows we shall call such an edge a *free* edge). We first show that a free edge must eventually be placed on some vertex's A-list.

So suppose no free edge is on an A-list. Then all free edges must be on B-lists. Let e be a free edge, and let it be the first such on the B-list for some vertex v. Let m be the number of edges in the concatenation of v's A- and B-lists that are in front of e. By assumption, the other endpoints of all these edges either are dead or have no free ports (and never will have, since no more edges will ever finish). Thus the response to any query concerning such an edge must be "No," which results in the edge being moved to the end of the B-list and the reduction of m by 1. Each time such an answer is received, we also reduce $N_{QUERIED}$ by 1, so that in Step 2 a new query is posed, either for e or some edge ahead of e in the concatenation of the A- and B-lists. Since all queries receive answers, and all edges ahead of e must receive the answer "No," this means that eventually the number of edges ahead of e will be reduced to the point where a query must be posed for e. The other endpoint of e, being neither busy nor dead, must thus answer "Later" (it can't answer "Yes" as that would cause e to be started), which causes v to put e on its A-list.

So now let us suppose that there are free edges on A-lists. Let e be the greatest free edge (under $>_G$) having this property, and let u and v be its endpoints, with at least v having e on its A-list. We shall show that either e gets started, or else some other free edge e' with $e' >_G e$ is promoted to an A-list. Suppose not, and hence e remains the greatest free edge, in perpetuity. By an argument like that of the previous paragraph, we can assume that e eventually reaches the head of v's A-list (after all non-free edges have received "No" answers and been demoted). Since a request for e can never receive a "No" answer, it will never leave its A-list. Assuming no other edge e' is promoted, e must thus remain at the head of its A-list in perpetuity. At some point it thus must be requested, and thereafter, each time Step 2 of the protocol is reached, e will either already be under request or else will be re-requested (this happens when a "Later" response was received in Step 1). Hence, from some point on, e will always be under request when Step 1 is entered. If, from this point on, v ever reads a request $Q(u,e,v)$ off its Protocol Message Queue in Step 1, it must answer "Yes" and start a transfer. Thus e's other endpoint u must never query for e. However, u must answer v's queries, and it must answer them with "Later," thus insuring that e gets on u's A-list. As soon as this happens it is only a matter of time before u must query for e, since no edge ahead of e on u's A-list can be free by our choice of e.

Thus we have a contradiction to our assumption that e is never started nor superseded as "greatest" free edge on an A-list. Hence one of these two possibilities must occur. Since the replacement of the greatest free edge can only occur a finite number of times (there are only a finite number of edges and by assumption no edge ever loses its freedom), eventually an edge must start, the final contradiction, from which the theorem follows. ■

As a consequence of Theorem 10, we can conclude that, so long as dead vertices are not revived and our assumptions about the underlying protocols are valid, protocol DP2 guarantees that all files involving surviving vertices will eventually be transferred successfully. Moreover, a statement analogous to Corollary 9.1 can be made. Let us assume that no vertex ever dies, no transfer fails, and that all edges are known to their endpoints at time 0. Then Theorem 10 says that any time an edge becomes free there is a finite amount of time before either an edge starts or another edge finishes. Let $\delta(DP2,G)$ be the maximum amount of time for this to occur given the network corresponding to the file transfer graph G.

Corollary 10.1. Assuming that all edges are known to their endpoints at time 0 and that no vertex ever dies and no transfer ever fails, then if s is the schedule produced by Demand Protocol 2 with *ex post facto* message graph G, e is the last edge finished, and p is the maximum port capacity, we have

$$MAKESPAN(s) \leqslant 2OPT(G) + L(e)\left[1 - \frac{1}{2p}\right] + 2\delta(DP2,G)\cdot|E|$$

$$\leqslant 3OPT(G) + 2\delta(DP2,G)\cdot|E|\,.$$

Note that the delay term includes a factor of 2 since we must include $\delta(DP2,G)$ twice for each edge (once for its start and once for its finish), whereas in Corollary 9.1 we needed count $\delta(DP1,G)$ only once per edge. The delay term is probably even worse than twice that for DP1, however, since $\delta(DP2,G)$ does not directly correspond to $\delta(DP1,G)$. The model is different, involving many more protocol messages. Moreover, as hinted in our proof of Theorem 10, $\delta(DP2,G)$ may itself be proportional to $|E|$, since we may have to wait for all the free edges to be "promoted" before we actually get around to starting any of them. However, if $\delta(DP2,G)$ can be kept small relative to $OPT(G)$, then this protocol can be adapted to approximate the $2.5OPT(G)$ guarantee of Decreasing List Scheduling. The key is in an appropriate definition for $>_G$.

In order to make this definition, we shall impose just a few more (reasonable) restrictions on the model. We assume that each vertex v has a unique identification number $ID(v)$, that each copy e of a multi-edge has a distinct name $Name(e)$ (presumably the name of the corresponding file), and that both endpoints of an edge have identical estimates of its length. (We do not require that the actual *ex post facto* length agree precisely with this estimate, however.) Given these definitions it is straightforward to devise an ordering $>_G$ that will behave as desired and whose restriction to the edges involving a given vertex can be computed locally, so long as that vertex knows the ID's

of its neighbors. Let $e = \{v,u\}$ and $e' = \{v',u'\}$ be two distinct edges with $ID(v) > ID(u)$ and $ID(v') > ID(u')$. Then $e >_D e'$ if (a) $L(e) > L(e')$, or if (b) $L(e) = L(e')$ and $ID(v) > ID(v')$, or if (c) $L(e) = L(e')$, $v = v'$, and $ID(u) > ID(u')$, or if (d) $L(e) = L(e')$, $v = v'$, $u = u'$, and $Name(e)$ is lexicographically prior to $Name(e')$.

Corollary 10.2. Suppose that the situation is as described in Corollary 10.1, that no actual edge length differs from the edge's estimated length by more than ϵ, and that the order relation $>_G$ used in DP2 is $>_D$. Then we have

$$MAKESPAN(s) \leq \frac{5}{2}OPT(G) + 2\epsilon + 2\delta(DP2,G)\cdot|E| .$$

Proof. Given Corollary 10.1, the result will follow if we can show that the last edge e to finish has actual length no more than $\frac{1}{2}OPT(G) + 2\epsilon$. Suppose not. Then the estimated length of e exceeds $\frac{1}{2}OPT(G) + \epsilon$, and any edge e' with $e' >_D e$ must have actual length exceeding $\frac{1}{2}OPT(G)$. We may also assume that e started after time $2\delta(DP2,G)\cdot|E|$, since otherwise the result would follow from the fact that the actual length of e is at most $OPT(G)$. Let us examine what happened at time 0 and immediately thereafter. Initially e was on the A-list for both its endpoints u and v. We may assume that it was among the first p_v edges on the A-list for v, as otherwise v would have had $p_v + 1$ edges of length exceeding $\frac{1}{2}OPT(G)$, which is impossible. Similarly, e must have been among the first p_u edges on the A-list for u. So consider the first time v executed Step 1 of DP2. If the Protocol Message Queue was not empty and the first message was a query from u concerning e, v would have had to say "Yes" as $N_{FREEPORT} = p_v$ and $N_{QUERIED} = 0$. Otherwise, at Step 2 v must have queried u about e. Turning now to u, we note that u must also either have said "Yes" to a query about e the first time it entered Step 1, or sent a query itself about e at Step 2. If neither u nor v says "Yes" to a query about e during its first pass through the protocol's main *while* loop, then both will have sent a query about e. One of the two must thus receive the other's query before it receives the answer to its own. Suppose without loss of generality that this happens to v. By Step 1.4.2 of DP2, v must then say "Yes." We conclude that one of u and v must say "Yes" to e, and hence the transfer must take place, without a query about e ever receiving a "No" answer. In other words, e remains free from time 0 until the time it is transferred. Thus by Theorem 10 and the definition of $\delta(DP2,G)$, e must start no later than time $2\delta(DP2,G)\cdot|E|$, a contradiction.

We thus can conclude that the actual length of the last task to finish is at most $\frac{1}{2}OPT(G) + 2\epsilon$, and the Corollary follows. ■

Note that the proof of Corollary 10.2 depends heavily on the fact that all vertices start executing DP2 at the same time (time 0). Otherwise u or v might have received a "No" answer from the other endpoint because it was not yet awake. This would wreak havoc with the argument of Corollary 10.2, although it would have only a limited effect on Theorem 10 and Corollary 10.1, as long as the spread between vertex start-up times is itself limited.

5. DIRECTIONS FOR FURTHER STUDY

In this paper we have introduced the problem of File Transfer Scheduling. As far as we know, there has been little previous theoretical analysis of this problem (although a more-limited variant, posed in terms of a link-testing problem, has been discussed in [4]). There are a wide variety of directions for further research, such as improving on our results, extending them to more general versions of the model, and investigating other optimization criteria.

Within the model, the major question we see is whether a polynomial time heuristic can be devised that will provide a better guarantee than the $\frac{5}{2}OPT(G)$ of Decreasing List Scheduling. (For those interested in the arcana of NP-completeness, there is also the question of the complexity of FILE TRANSFER SCHEDULING for message graphs that are odd cycles of equal-length multi-edges.)

The most important extension of the model is that which includes the possibility of *forwarding*. This becomes relevant if u wants to send a file to v, but has no direct link, and hence must send it to an intermediary, say w, who will then send it on to v. Forwarding may be helpful even when direct links are available. Suppose both u and v are so busy that times when they simultaneously have free ports are rare. In this case a common neighbor w, which has a free port most of the time, might take the file from u when u has the time and then hold it until v is free. Can our results be extended to include one or both of these types of forwarding?

Another extension of the model would be to include the real-life distinction that often arises between dial-out and dial-in ports, as in the case of a computer system with just one automatic call unit, but many incoming lines. In such a situation it might be that two nodes each have many ports, but can only be involved jointly in two simultaneous file transfers, since each transfer occupies one of the dial-out lines.

Finally, there is the question alluded to earlier about appropriate optimization criteria. Makespan is a reasonable standard in the case when one is trying to perform all transfers during a particular time interval, or when all the nodes are going to be tied up in the transfer process until all transfers are completed. However, in many systems, file transfers may be going on all the time, or nodes may be free to do other things once their own transfers are complete. The latter case suggests the relevance of a criterion such as "average finishing time" (over all nodes), the former a more dynamic measure such as the "average time in system" for a requested transfer, i.e., the average delay between the time a node decides it wants the transfer until the transfer is accomplished. For such criteria, average case analysis may well be more meaningful than the worst case analysis performed here, especially if one wishes to analyze the system when nodes or transfers may fail.

REFERENCES

1. C. BERGE, "Graphs and Hypergraphs," North-Holland Publishing Company, Amsterdam, 1973.
2. E. G. COFFMAN, JR, M. R. GAREY, AND D. S. JOHNSON, An application of bin-packing to multiprocessor scheduling, *SIAM J. Comput.* 7 (1978), 1-17.
3. R. COLE AND J. HOPCROFT, On edge coloring bipartite graphs, *SIAM J. Comput.* 11 (1982), 540-546.
4. S. EVEN, O. GOLDREICH, AND P. TONG, "On the NP-completeness of certain network-testing problems," Report No. 230, Department of Computer Science, Technion, Haifa, Israel, 1981.
5. D. K. FRIESEN, Tighter bounds for the MULTIFIT processor scheduling algorithm, *SIAM J. Comput.*, to appear.
6. H. N. GABOW, private communication (1982).
7. H. N. GABOW AND O. KARIV, Algorithms for edge coloring bipartite graphs and multigraphs, *SIAM J. Comput.* 11 (1982), 117-129.
8. M. R. GAREY AND D. S. JOHNSON, "Computers and Intractability: A Guide to the Theory of NP-Completeness," W. H. Freeman and Company, San Francisco, 1979.
9. I. HOLYER, The NP-completeness of edge-coloring, *SIAM J. Comput.* 10 (1981), 718-720.
10. S. SAHNI, Algorithms for scheduling independent tasks, *J. Assoc. Comput. Mach.* 23 (1976), 116-127.
11. C. E. SHANNON, A theorem on colouring the lines of a network, *J. Math. Phys.* 28 (1949), 148-151.
12. V. G. VIZING, On an estimate of the chromatic class of a p-graph (in Russian), *Diskret. Analiz.* 3 (1964), 25-30.

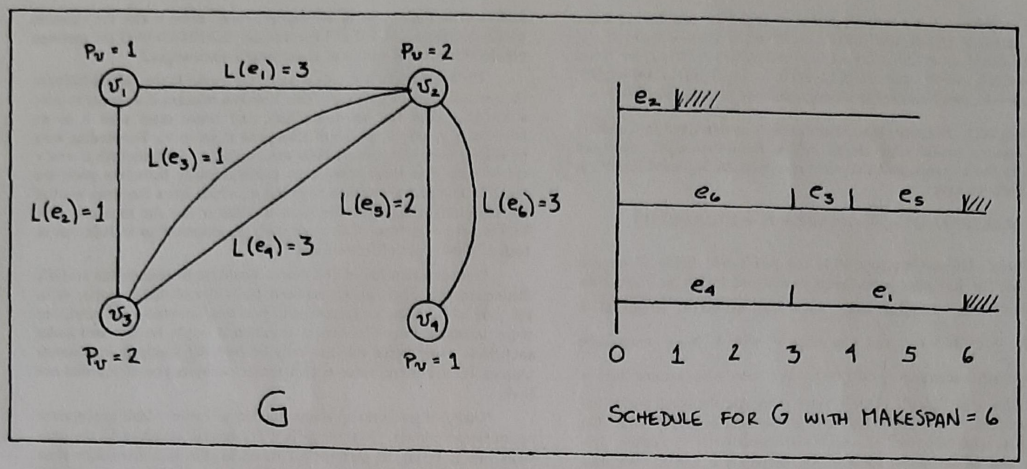

FIGURE 1. AN EXAMPLE FILE TRANSFER GRAPH AND SCHEDULE.

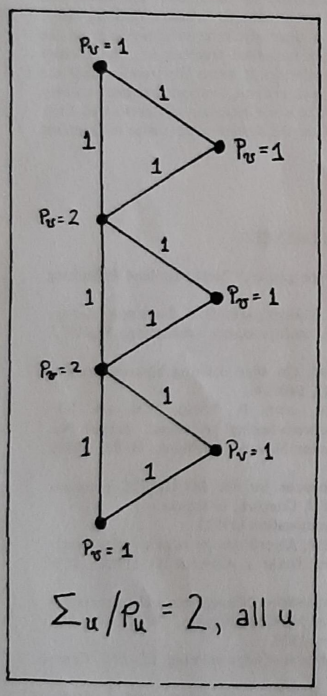

FIGURE 2. AN EXAMPLE WHERE OPT(G) > $\max_u \{\Sigma_u / P_u\}$.

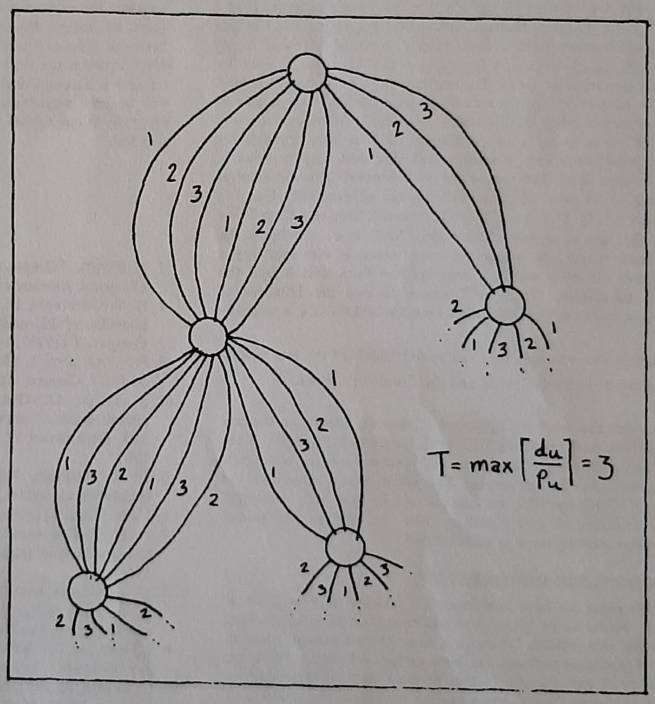

FIGURE 3. THE EDGE LABELLING USED IN THEOREM 4.

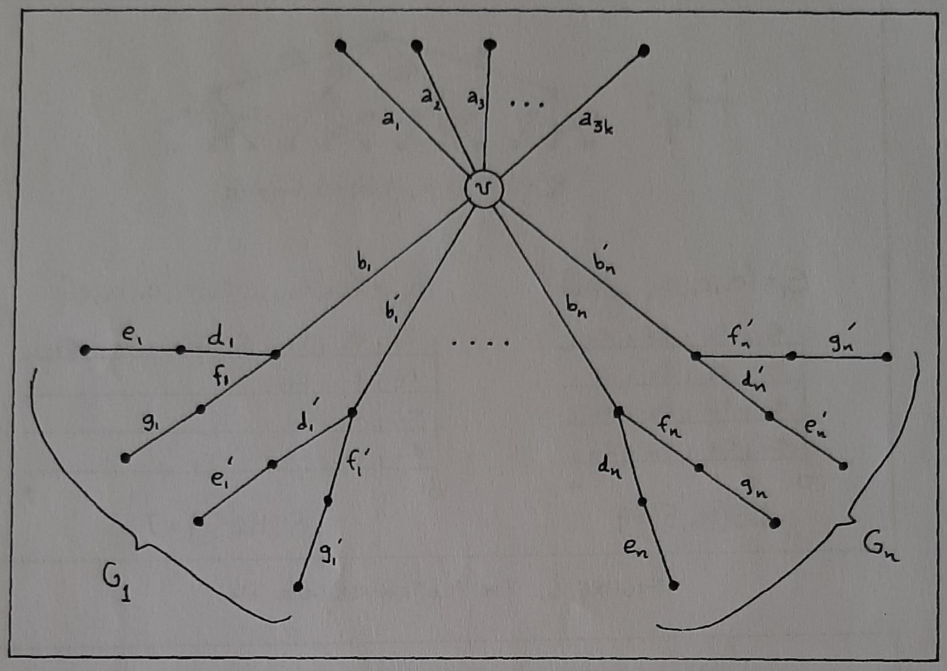

FIGURE 4. TREE USED IN PROOF OF THEOREM 5.

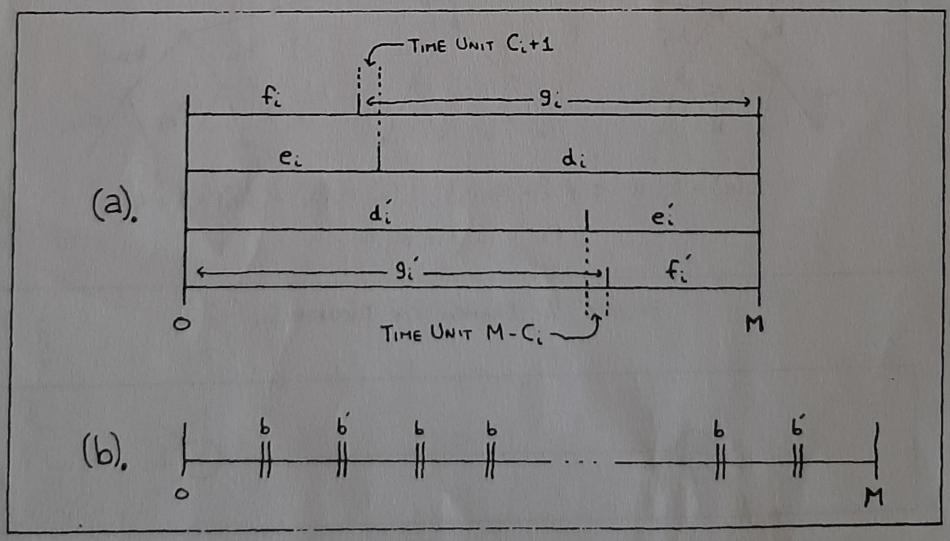

FIGURE 5. SCHEDULING THE TREE OF FIG. 4

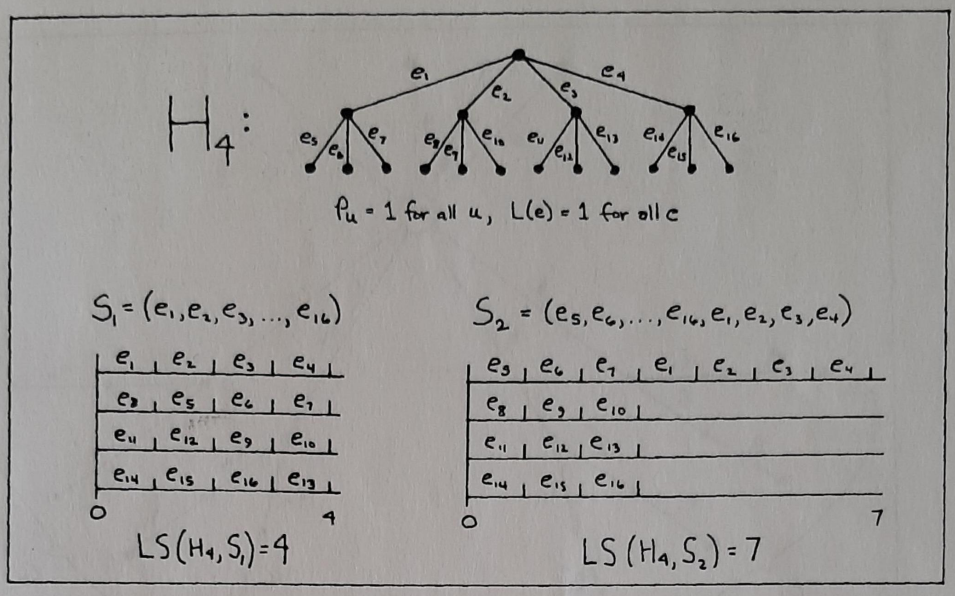

FIGURE 6. THE FILE TRANSFER GRAPH H_4.

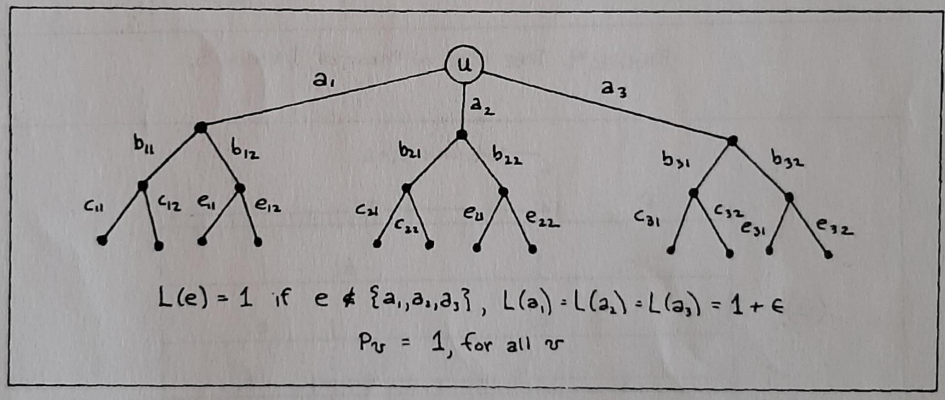

FIGURE 7. EXAMPLE FOR THEOREM 6.

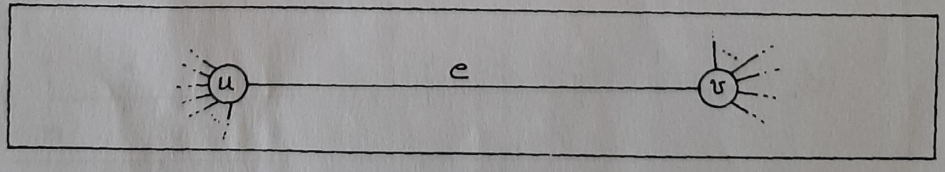

FIGURE 8. ILLUSTRATION FOR THEOREM 7.