MCM 1994 Problem B
Network File Transfer Scheduling

1. Restatement of the Problem

Our company has a network of computers which must share information (contained in files) on a regular basis. It is known which computers must share with others, and how long each "sharing" (file transfer) takes. Also, some computers might have the ability to carry out more than one transfer at a time.

Our task is to schedule the file transfers in such a way that the total time required is a minimum. This minimum total time is called the "makespan" of the network.

2. Assumptions

Rational Time: Any time expressed in this problem or its solution will be rational. This is because it is impossible to measure an irrational time in the real world.

Discrete Time: Because all times are rational, any time in this problem or its solution may be expressed as an integer multiple of some fundamental time quantum. This quantum may be taken as the largest value such that all transfer times are integer multiples of it.

Bidirectionality: In any given file transfer, it is not significant which computer is sending or receiving. As long as a computer is doing either, it is listed as "involved" in that transfer.

Negligible Overhead: The times as listed are for the actual transfer of data. The time it takes to start/stop a file transfer is negligible. Therefore, if a computer is involved in a transfer from time 0 to 1, and a different transfer from time 1 to time 2, there is no conflict at time 1.

Transfer Continuity: Once a transfer is started, it must run to completion. That is, a file of length 2 may not be transferred from 0 to 1, stop, and finish from 3 to 4. The negation of this assumption is discussed in Appendix B.

Physical Connectivity: The diagrams given indicate only necessity and possibility of file transfers. They do not contain any other information about the physical setup of the network.

3. Motivation for the Model

Any situation in which nodes (computers) must be connected to each other reminds one immediately of graph theory. This is true particularly when each connection must have a value associated with it, like weighted edges on a graph. The assumption of Bidirectionality implies that one might as well forget about file transfer directions altogether. Therefore, an undirected, weighted graph would be appropriate.

4. Design of the Model

Because nothing is known of the physical setup of the network, the undirected weighted graph consists of the necessity graph for the network. That is, each node in the graph corresponds to one and only one computer. Each edge connecting two nodes corresponds to exactly one necessary file transfer, and the weight on that edge is the time required for that transfer, in time quanta (forcing the weights to be integers).

The N nodes are enumerated 1. . .N, and node i is denoted by V_i. The number of simultaneous transfers node i can do is denoted by T_i. The weight on an edge between nodes i and j is denoted $e_{i,j}$ for all i and j (Note that this looks similar to, but is different than, the notation in the official problem statement.) The value of $e_{i,j}$ is defined as follows:

$$e_{i,j} = \begin{cases} \text{time to transfer file between } V_i \text{ and } V_j & i \neq j, V_i \text{ connected to } V_j \\ 0 & i \neq j, V_i \text{ not connected to } V_j \\ 0 & i = j \end{cases}$$

A schedule is a function from an ordered 3-tuple of integers into the set {true, false}. It is defined as follows:

$$S: (Z_N, Z_N, Z) \rightarrow \{\text{true}, \text{false}\}$$

$$S(i,j,t) = \begin{cases} \text{true} & \text{if a file is being transferred between } V_i \text{ and } V_j \text{ at time } t \\ \text{false} & \text{otherwise} \end{cases}$$

In this way, a schedule may be represented as a 3-dimensional array of Booleans, with one dimension (time) infinite. A somewhat smaller way to represent a schedule is as a set of ordered integer 4-tuples ($start\_time, end\_time, node\_a, node\_b$), which effectively says that the edge connecting $node\_a$ to $node\_b$ will be transferring a file from time $start\_time$ to end_time.

A solution is defined as a valid schedule. That is, a solution successfully transfers all files, and no node is ever overloaded. If we let $\tau(s) \equiv$ total time for solution s to complete, then makespan $\equiv min_{\forall \text{solutions}_s} \tau(s)$. The set of all s that achieve this minimum is the set of optimal solutions.

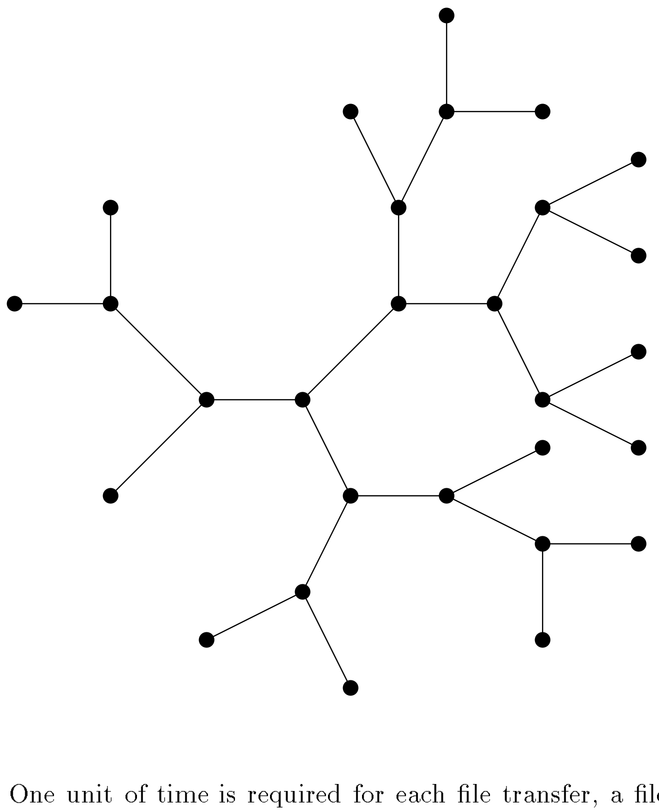
Please see Appendix C for properties and definitions related to this model.

5. Testing the Model

In the problem statement, three situations were presented to apply the model to. Their solutions are presented here. In all cases, the numbers of the vertices are arbitrary, so they were omitted.

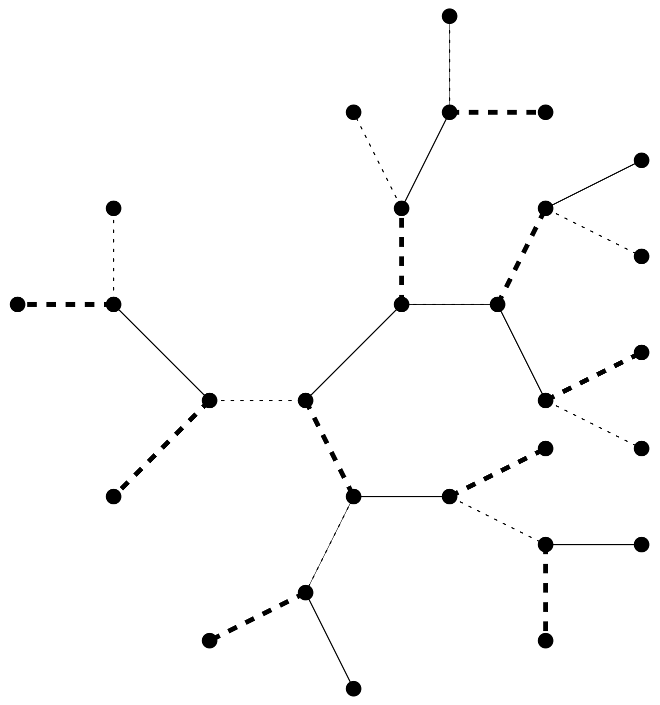
Situation A

The following graph represents the systems under consideration and the transfers that must be made between them. Each of the 28 nodes represents a departmental computer, and each of the 27 edges represents a file transfer that must be made daily.



One unit of time is required for each file transfer, a file transfer involves both sender and receiver, and a computer can only be involved in one transfer at a time.

Some of the nodes in this graph have degree three; hence, they will be involved in three file transfers. Since each transfer requires one unit of time, these nodes will require at least three units of time to complete all the necessary transfers. Consider the three sets of edges in the graph represented by the solid, dashed, and dotted edges in this representation of the graph:



Note that no two edges of the same type are incident upon a single node. Hence, all the file transfers represented by the solid edges can be performed simultaneously, as can those represented by the dashed edges and those represented by the dotted edges. Performing the transfers along the solid edges, followed by the transfers along the dashed edges, and finally those along the dotted edges will require three units of time. As we have seen, at least this much time is required to perform the transfers. Hence, performing the transfers in this manner is optimal. The makespan is therefore three.

Our approach to this problem takes advantage of several properties of this special case. The graph is a tree with maximum degree three. Since each transfer takes 1 unit of time, we do not need to worry about transfers overlapping in time. They are either simultaneous or completely disjoint. Also, each computer can only handle one transfer at a time. So during each interval of time, the active edges represent a matching of nodes, that is, they are vertex disjoint copies of K_2 (the complete graph on 2 vertices). Because this tree is of maximum degree 3, such a matching is easy to find. Starting with any leaf, we traverse the tree. Whenever we come to a vertex not involved in a transfer, we select an adjacent vertex that has not yet been visited. We add the edge between these two vertices to the set of edges which will execute on the first step. This traversal guarantees that all the nodes not involved in transfers are leaves.

After the first time interval, we will be left with only paths. By choosing alternating edges we can finish in 2 more units of time.

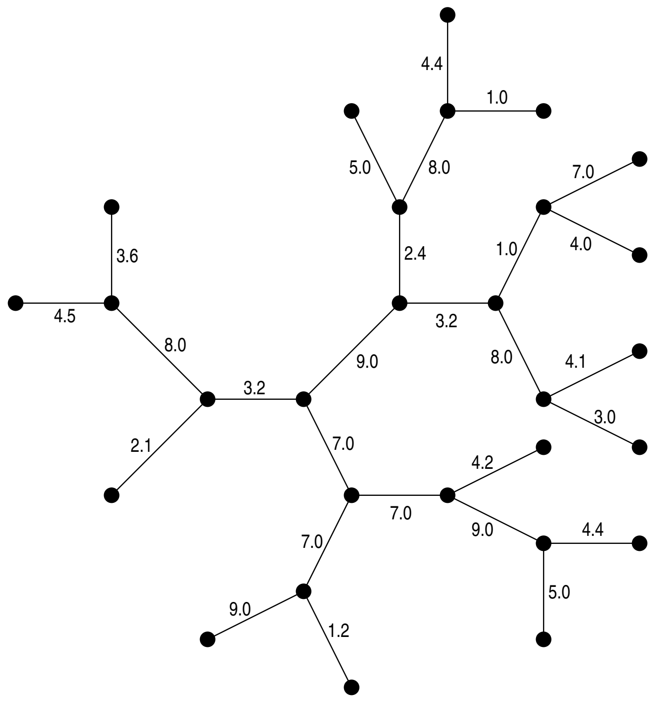
This approach will work on any tree with maximum degree 3 for which transfers require one unit of time and for which each computer can only handle one transfer at a time. So a tree with these properties will have a makespan of 3.

From this analysis, we can see that there are clearly multiple ways to choose the individual edges, since there will be many arbitrary choices regarding which edge to include. Hence, the given schedule is not a unique solution.

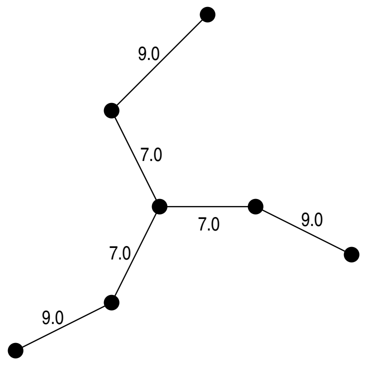
This approach depends heavily on certain aspects of the problem, and therefore will not work in the general case in which the graph, transfer times, and node capacities are arbitrary. Indeed, finding the makespan for the general case will prove problematic, since the general problem is NP-complete. (See Appendix A for a discussion of this.)

Situation B

Consider the graph from Situation A, but with the required transfer times as indicated below:

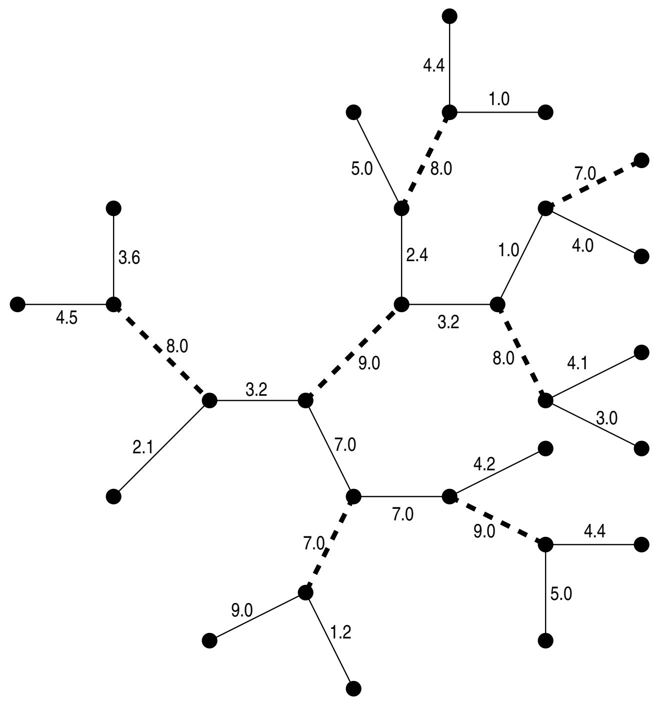


Note that this contains the graph

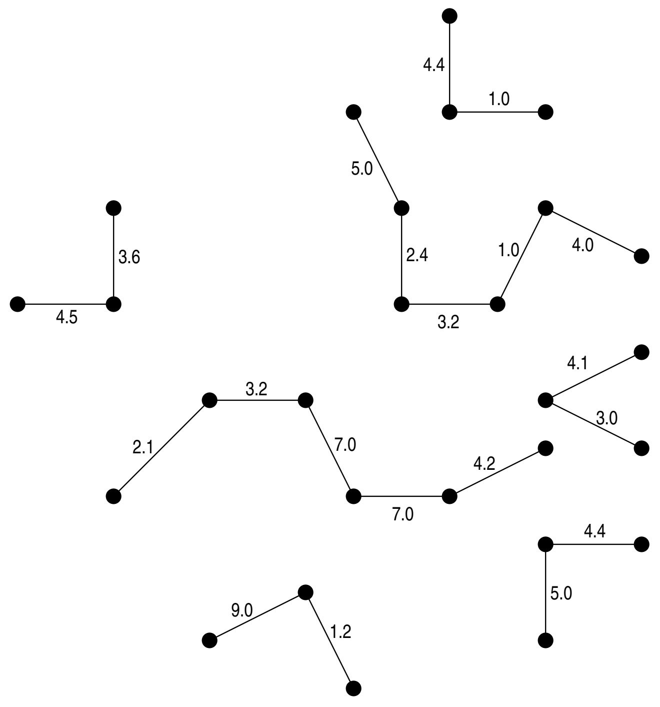


The presence of the three transfer times of 7.0 around the central node of this graph implies that 21.0 units of time is a lower bound on the amount of time required to perform all the transfers. Consider the edge with value 7.0 that is handled second. At least 7.0 units of time are required both before and after it to handle the two other 7.0-valued edges. Furthermore, since it has a vertex in common with an edge of value 9.0, 9.0 units of time will be required either before or after it. Hence, at least $7.0 + 7.0 + 9.0 = 23.0$ units of time will be required, regardless of whether the 9.0 is transferred before or after the second 7.0. So this graph requires at least 23.0 units of time.

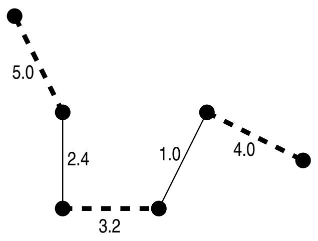
Now consider the effect of starting with the transfers indicated here by dashed lines, and letting all these transfers run to completion (which takes 9.0 units of time):



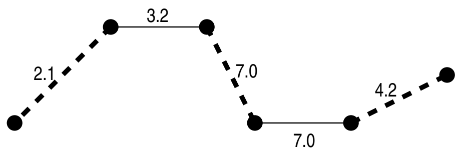
When all these transfers are complete, we will have the following transfers remaining to perform:



The time required to finish the transfers at this point will be the maximum of the times required to handle the individual subgraphs that remain. Note that all of the remaining subgraphs with two edges can be finished in 10.2 units of time or less. The path



Can be handled in $5.0 + 2.4 = 7.4$ units of time by performing the transfers indicated here by dashed lines, then those indicated by solid lines. The final path



Can be handled in $7.0 + 7.0 = 14.0$ units of time by performing the transfers indicated by dashed lines followed by those indicated by solid lines.

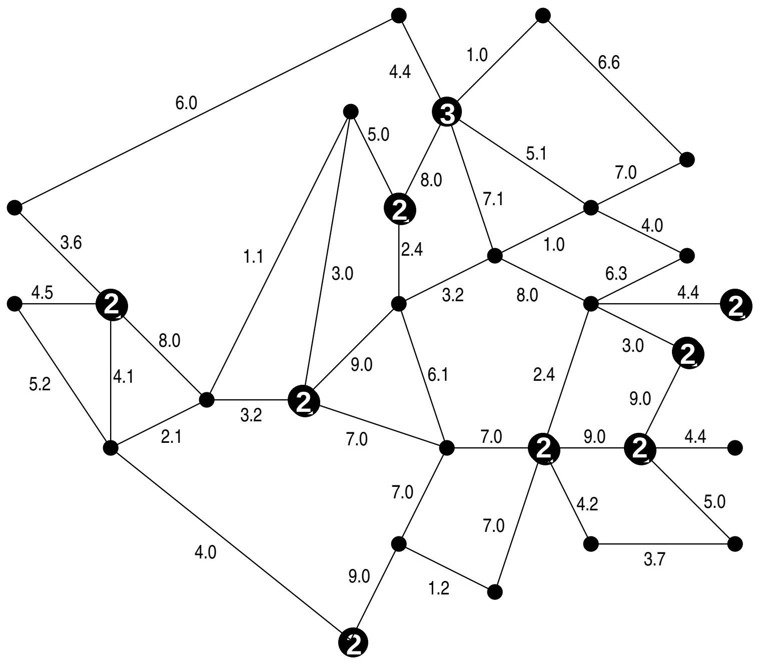
So the transfers that remain after we handle the first set specified will take 14.0 units of time to complete. Since the first set required 9.0 units of time, performing all the transfers in the manner described will require $9.0 + 14.0 = 23.0$ units of time to complete. Since we have seen that at least 23.0 units of time are required, this solution is optimal. So the makespan is 23.0

This problem was solved in a similar manner to the first problem, in that a maximal matching was chosen initially. However, the assumptions about execution time from Situation A are no longer valid. So, to some extent this solution comes from the "guess and hope" method—the initial matching was chose to quickly get many large transfers out of the way. It then turned out that the remaining pieces could be handled in a manner which achieved a value which was known to be a lower bound. To obtain this lower bound, it was important to notice that the obvious lower bound of 21.0 units of time was unattainable because of the arrangement and values of some of the edges.

The general case is naturally still NP-complete, so this approach will certainly not work in general. In fact, this graph falls into a more specific case, that where the graph is a tree and each node has capacity 1, which is also NP-complete despite the additional restrictions. (See Appendix A.)

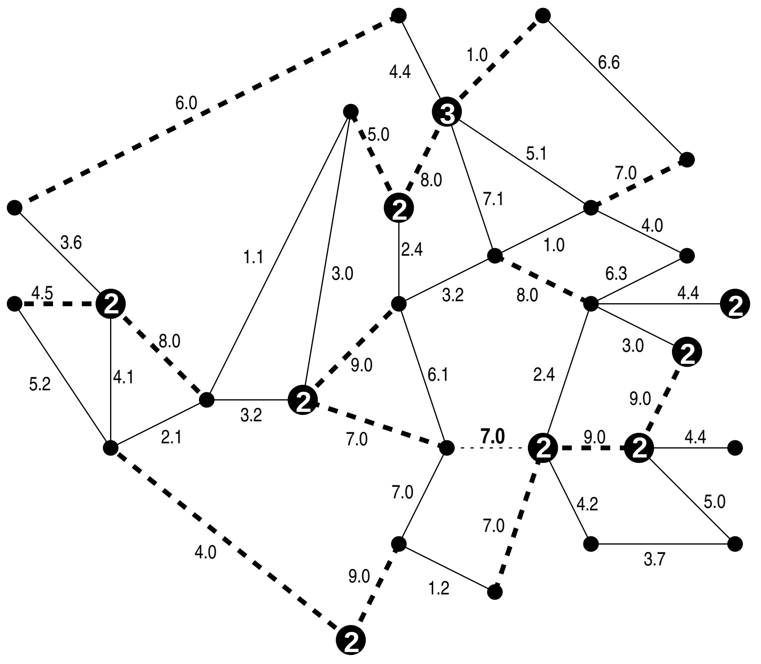
Situation C

We now have the following graph, which happens to contain the graph from situation B. Also, some nodes are now capable of transferring more than one file at a time. Nodes with this ability are shown here with the number of simultaneous transfers they can handle.

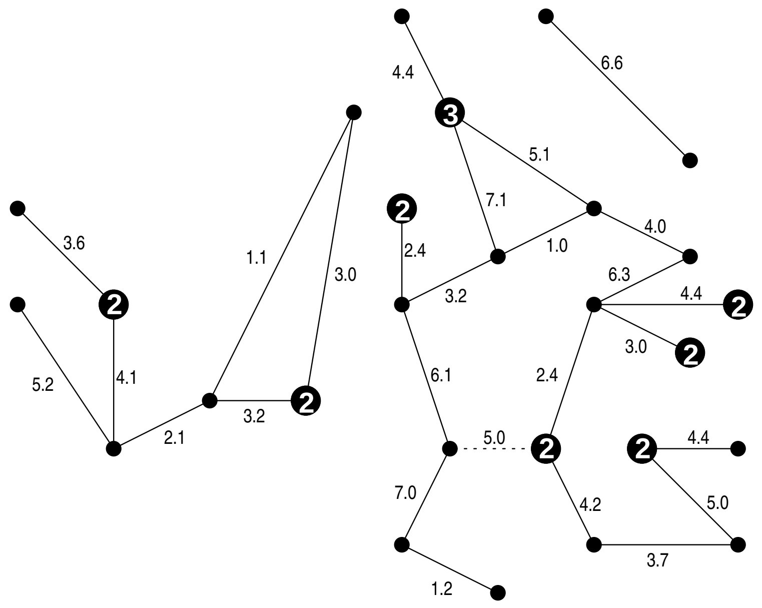


One of the nodes in this graph has to make three transfers which require 7.0 units of time, and one which requires 6.1 units of time. Hence, a total of 27.1 units of time are required for this node to finish its transfers. It follows that 27.1 is a lower bound on the time required to finish the transfers indicated in this graph.

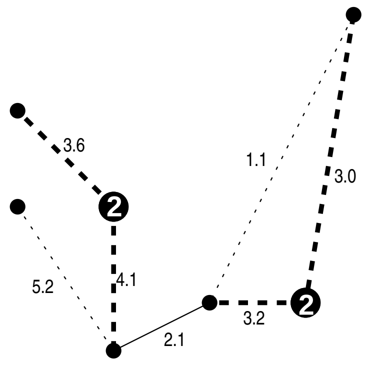
Now suppose that we begin by running the transfers indicated with dashed lines, and letting these run until they are all complete (requiring 9.0 units of time). In addition, let the transfer indicated with a dotted line begin after 7.0 units of time have elapsed.



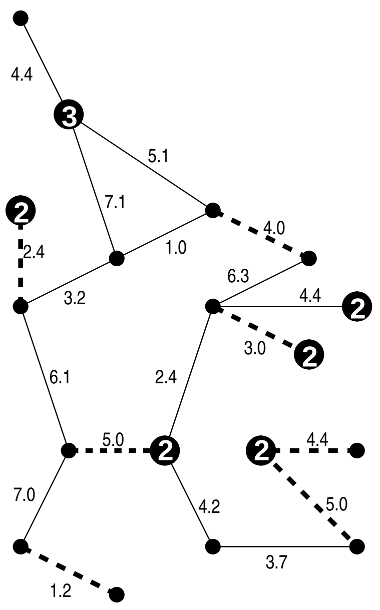
When 9.0 units of time have elapsed, we have the following graph, with the constraint that the transfer shown by the dotted line must "begin" executing immediately, since it is really the continuation of a transfer started earlier and therefore cannot be interrupted:



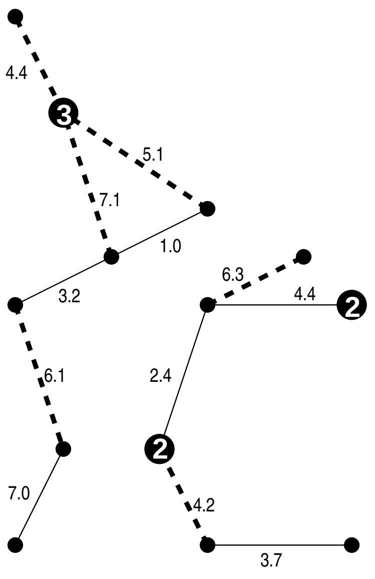
The isolated edge can then be finished in 6.6 units of time. The left component can be finished in 11.4 units of time by first running the edges indicated here with dashed lines (requiring 4.1 units of time), then those with dotted lines (requiring 5.2 units of time), and finally the solid edge (requiring 2.1 units of time):



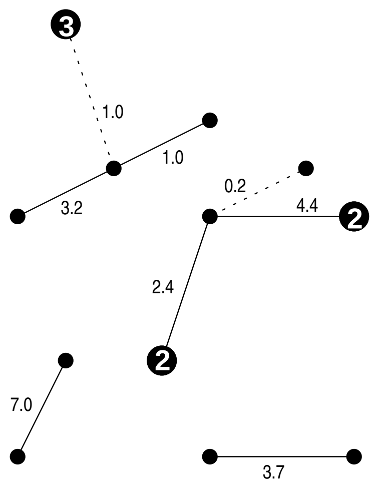
To handle the large component, let the transfers indicated below with dashed lines run to completion (note that this obeys our requirement that the transfer already in progress continue running):



So after 5.0 units of time beyond the initial 9.0, we are left with the following graph:



Let the transfers indicated by the dashed edges above run to completion. Consider the graph after 6.1 units of time have elapsed. We then have the following, where again the dotted edges must be running initially, since they represent the continuation of transfers:



Both isolated edges above can be completed in 7.0 units of time or less. The other two remaining graphs can be handled in $1.0 + 1.0 + 3.2 = 5.2$ and $0.2 + 4.4 + 2.4 = 7.0$ units of time. Everything that remains at this point can therefore be completed in 7.0 units of time. So handling the large component that remains after the initial round of transfers requires $5.0 + 6.1 + 7.0 = 18.1$ units of time. This is the largest of 6.6, 11.4, and 18.1, so to complete everything that remains after 9.0 units of time have elapsed will require 18.1 units of time. Hence, the total time required to perform these operations is $9.0 + 18.1 = 27.1$ units. Since we have seen that this is a lower bound on the time required to perform the transfers, the given schedule is optimal.

Our solution for this problem was found in essentially the same manner as that for Situation B, namely, start guessing and see if it works. Though this graph is more formidable, it is helpful that the obvious lower bound of 27.1 can be achieved. The only obvious category for this problem is the most general case, which is NP-complete. Hence, it is extremely unlikely that any efficient algorithm exists for producing solutions to scheduling problems of this variety.

6. Strengths and Weaknesses of the Model

Graph models are usually easy to program, because so much work has been done with graph theory.

The model allows integer arithmetic for almost everything, due to the time quantum. This almost eliminates any roundoff errors (some may happen during efficiency calculations), and does eliminate iterative error growth.

The assumption of Transfer Continuity is a weakness in that it limits the makespan of some graphs. Appendix B deals with what can happen without it.

The assumption of Physical Connectivity allow the model to completely ignore the underlying hardware. This can be seen as either a strength (less processor time) or weakness (no error checking), depending on the reliability of the network.

Appendix A: NP-completeness of Makespan Determination

Consider the following special case of the general problem of finding the makespan for an arbitrary network:

> Let G be a tree representing a network, with times $T_1, T_2, \ldots, T_k$ assigned to its k edges. If we interpret each edge to represent the time required for a file transfer between two nodes, what is the minimum time in which all the transfers can be performed, given that no node can be involved in more than one transfer at once, and that transfers may not be interrupted and resumed later?

This is closely related to the following decision problem:

> Let G be a tree representing a network, with times $T_1, T_2, \ldots, T_k$ assigned to its k edges. Let n be a non-negative integer. If we interpret each edge to represent the time required for a file transfer between two nodes, can all the transfers be performed in time n or less, given that no node can be involved in more than one transfer at once, and that transfers may not be interrupted and resumed later?

If we can solve this decision problem, we can simply try it for each successive value of n until an answer of "yes" is obtained. We are guaranteed to eventually reach such an answer, since no graph can require more than $\sum T_i$ units of time to complete its transfers.

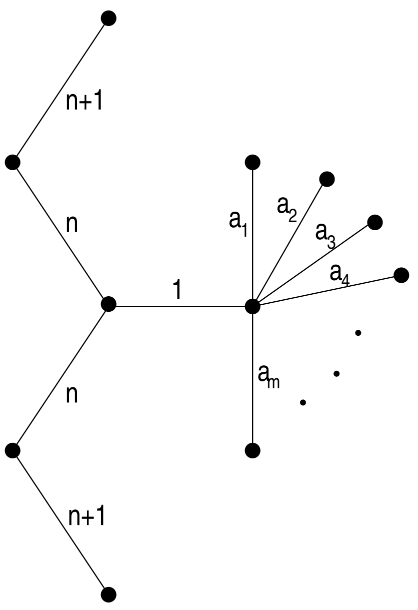
Theorem: *The decision problem given above is NP-complete.*

Proof: We will prove this by reducing from the partition problem, which is:

> Given integers $a_1, a_2, \ldots, a_m$, is there a partition of these integers into sets A and B so that $\sum_{a_i \in A} a_i = \sum_{a_i \in B} a_i$?

The partition problem is NP-complete (Garey and Johnson 60).

Let an instance of the partition problem be given, with integers $a_1, a_2, \ldots, a_m$. If $\sum a_i$ is odd, then there cannot be a partition into two sets with equal sum. So suppose $\sum a_i$ is even. Take n so that $2n = \sum a_i$. Now construct a tree with the following structure and edge weights:



Suppose the transfers indicated in this graph can be performed in $2n+1$ units of time. This implies that the two nodes with edges of values n and $n+1$ will spend the entire time engaged in transfers. So they can each either perform the n unit transfer followed immediately by the $n+1$, or the $n+1$ followed by the n. If they both perform the $n+1$ first, then the n unit transfers will not be possible at the same time, because they are incident upon a common node. This means that at least $(n+1)+n+n = 3n+1$ units of time are required, which contradicts our supposition that the transfers can be performed in $2n+1$ units of time. Similarly, if both n unit transfers are performed first, the $n+1$ unit transfers will not be possible at the same time, and $n + (n+1) + (n+1) = 3n+2$ units of time will be necessary. Hence, one of the $n+1$'s must be earlier, and the other must be later. This means that the only time that the 1 unit transfer can be performed is in the one unit of free time between the two n's.

It then follows that the node connecting the a_i's is occupied from time n to $n+1$ (assuming the clock starts at time 0). Since $\sum a_i$ is $2n$, this node must also be continually engaged in transfers. So the durations of the transfers it performs before handling the 1 unit transfer must sum to n, as must the durations for those performed after the 1 unit transfer. Let A be the set of the durations for the transfers handled in the first half; let B be the set of the durations for the transfers handled in the last half. A and B constitute a partition of $a_1, a_2, \ldots, a_m$, and

$$\sum_{a_i \in A} a_i = n = \sum_{a_i \in B} a_i$$

So if this graph can perform its transfers in $2n+1$ units of time, there is a partition of $a_1, a_2, \ldots, a_m$ such that $\sum_{a_i \in A} a_i = \sum_{a_i \in B} a_i$.

Now suppose that there is such a partition into sets A and B. As before, if the 1 unit transfer is run at the halfway point we can run the two $n+1$ transfers, the two n unit transfers, and the 1 unit transfer in a total of $2n+1$ units of time. The node joining the a_i's is then available for n units of time before the 1 transfer, and n units of time afterward. We can handle the transfers corresponding to the elements of A in the first n-unit interval, and those corresponding to B in the last n-unit interval. So if there is a partition of $a_1, a_2, \ldots, a_m$ into A and B such that $\sum_{a_i \in A} a_i = \sum_{a_i \in B} a_i$, then the corresponding graph can perform its transfers in $2n+1$ units of time.

Hence, the desired partition exists if and only if the corresponding graph can be executed in $2n+1$ units of time.

Given an instance of the partition problem, the time required to produce a description of the corresponding graph is clearly of polynomial order in the size of the input. So the partition problem is polynomial-time reducible to the graph scheduling problem. Also, if we are given an integer N and a proposed schedule for any graph, we can test in polynomial time whether it is a valid schedule requiring no more than time N by verifying that no node is ever involved in two transfers simultaneously, and checking if any transfers happen after time N. Hence, the graph scheduling problem described above is NP-complete. ∎

It then follows that any more general formulation of the scheduling problem is also NP-complete. In particular, allowing arbitrary graphs, multiple file transfer capability, or both results in an NP-complete problem.

References

Garey, Michael R. and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* New York: W.H. Freeman and Company, 1979.

Appendix B: Improving Network Performance

The assumption of Transfer Continuity has a very limiting effect on the makespan for many networks. In many cases, a node is idle for, say, 7 minutes, and has a file that takes 10 to transfer. This opportunity is lost, and the 10 minutes are tacked on the end of its task list. If files could be fragmented, though, only 3 minutes would be added. These time savings could occur at many nodes on the graph.

Important to this section is the assumption of Negligible Overhead. If fragmenting a file into two parts incurred twice as much overhead, performance could suffer.

It is clear that an algorithm that did not assume Transfer Continuity (henceforth called the Fragment algorithm) would produce a solution at least as good as one that did, because Fragment could always revert to a solution that happened to have no files fragmented.

Fragment would have the opportunity to disregard what previous time steps had done. For an example, the solution to Situation C, above, at one point had "the constraint that the transfer shown by the dotted line must "begin" executing immediately, since it is really the continuation of a transfer started earlier and therefore cannot be interrupted". Fragment could reject the continuation of this transfer if it determined that a different transfer would be more efficient for that node.

Here is a possible Fragment algorithm, in pseudocode:

```
While there are edges left
For each node in the graph compute an "urgency", such as the sum of the weights of all edges at that node.
While there are free nodes
Find the most urgent node, $V_i$
Find the edge to its most urgent, non-busy neighbor $V_j$
Add that edge to a list of active edges this iteration
Mark both nodes as having one more transfer active (in the case $T_i > 1$ or $T_j > 1$)
Keep track of how short the shortest edge is for this iteration
For each edge in the "active edges this iteration" list
Output part of a schedule saying that this edge was active at this interval
Decrease time left on edge by shortest time this iteration
```

The collection of the output schedules then forms a solution to the graph.

Appendix C: Consequences of the Model

Given an arbitrary graph that fits the model,

Property: $(\forall i, j \in Z_N)[e_{i,j} = e_{j,i}]$ This follows from Bidirectionality and the definition of $e_{i,j}$

Property: There exists a solution s. One may construct a solution in the following manner:

Enumerate the edges $1 \ldots M$. Run edge 1 to completion, with no others running. Then run edge 2, and so forth. This will complete every edge, and no collisions will happen. Note that this is not a very good solution.

Definition: If s is a solution, then the "time reversal" of s, denoted s^r, is as follows: if s is listed as 4-tuples (as suggested above), the i'th tuple is $(start_i, end_i, a_i, b_i)$. Create s^r by forming a list in which the i'th tuple is $(\tau(s) - end_i, \tau(s) - start_i, a_i, b_i)$.

Property: If s is a solution, then s^r is a solution. If two transfers didn't conflict, they won't conflict if done in reverse order. If all transfers completed, they will complete when done in reverse order.

Property: It is not necessarily true that $s = (s^r)^r$ If s has some time at the beginning when all nodes are idle, then upon time reversal, $\tau(s)$ will shrink by that amount of time. Subsequent reversal thus takes place in a smaller interval.

Definition: the extremely naïve lower bound on $\tau(s)$ is

$$\max_{i,j \in \{1 \ldots N\}} e_{i,j}$$

Obviously, the total time can be no smaller than the largest single edge time, since negative times are not allowed.

Property: There exists an optimal solution s^* Because time was quantized, any graph that fits the model has a makespan that is an integer times the time quantum. That is, effectively, an integer. Let $S \equiv \{s : sisasolution\}$. $S \neq \emptyset$ because there is a solution, as constructed above. Let $\tau(S) \equiv \{\tau(s) : s \in S\}$. Then $\tau(S)$ is a nonempty set of integers, bounded by the extremely naïve lower bound as above. Then $\tau(S)$ has a least element (by definition, the makespan). This least element has at least one solution s^* for which $\tau(s^*)$=makespan. This s^* is therefore an optimal solution.

Property: If s^* is an optimal solution, then $(s^*)^r$ is an optimal solution. Thus it is possible to have a nonunique optimum. By inspection, an optimal solution has no time when all nodes are idle. Therefore, $\tau(s^*) = \tau((s^*)^r)$ and $(s^*)^r$ is optimal.

Definition: the naïve lower bound on $\tau(s)$ is

$$\max_{i \in \{1 \ldots N\}} \frac{\sum_{j=1}^{N} e_{i,j}}{T_i}$$

The sum divided by T_i is a lower bound on the total time each node must spend active. The total time may be no smaller than the longest any node spends active. Property: if the graph is a tree, and is at most binary, then an upper bound on the makespan is three times the naïve lower bound. One can "worsen" the graph so all edges take as long as the longest one, the naïve lower bound. This equalizes all the times. One can also degrade all T_i to 1. The graph now fits Situation A, which has a known makespan of three times the constant edge length. Since this graph is more restricted that the original, the original makespan must be less than or equal to the new one. Note: this not only bounds the makespan, but easily produces a tree that achieves it.

Appendix D: Algorithm for situation B

No proof was put forth showing that the class of which situation B was an example was NP-complete. While the general case was shown to be NP-complete, this does not forclude that any given sub-class of the problem is also NP-complete. It also certainly does not restrict a human using poorly understood (intuitive) methods from coming up with a solution. The restrictions on the class from which situation B appears to be drawn are: binary tree form, that is every node on the graph has at most measure three and that there are no cycles in the graph, and each node can only participate in one file transfer. No further restriction is placed on the edge weight.

Since we do not know that there is not a good, that is polynomial time, algorithm for this class, one was attempted to be developed. The algorithm is split into two parts. One part determines which edges should be chosen initially. The second part propagates these through time and determines when to start the other edges.

The first part looks for a set of edges which can all be done initially and which get the most done as early and as compactly as possible. To do this, each edge has an "efficiency" measured for it. This value is the sum of all edges which can be run currently with the given edge, given by a skip-and-choose-largest (or "skippy") algorithm, divided by the maximum time for any edge in that set. The set having the highest efficiency measure is used for the initial edges to be run.

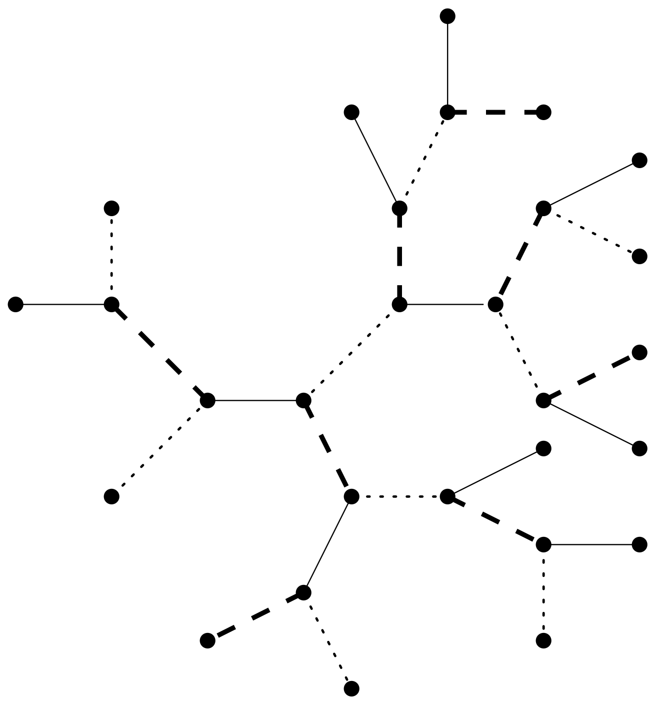
The "skippy" algorithm for picking a set of concurrent edges to be run initially rests on the assumption that an internal node should not be left idle initially. A graph was developed which contradicted this assumption, but it was decided to develop this algorithm anyway due to a lack of other alternatives. This algorithm is considered a greedy algorithm since it tries to pick the largest possible values first.

An edge is used as a seed for "skippy." This edge is guaranteed to be in the set of concurrent edges. From this edge, the set of nodes which are have transfers with either of the end nodes of the given edge is generated. All edges connecting the end nodes with any of the nodes in the set are eliminated from consideration of usable edges. The largest available edge off of each node in the set is determined. This algorithm is recursively called on that edge, being given the available edges, the used nodes, and a marker which keeps track of the largest edge. The returned result is added to a running sum of edges. When all nodes have been dealt with, the weight of the initial edge that was passed is added to the running sum and that total is returned. In the process of determining these values, the available edge and busy node matrix has been updated.

Since the function is recursive, it is up to the original caller to do the division by the largest value. Also, the available edges at the end is the set of edges which was chosen. The set of edges which give the largest efficiency is retained and used as the initial edges. Once the initial edges have been determined the propagating algorithm is employed.

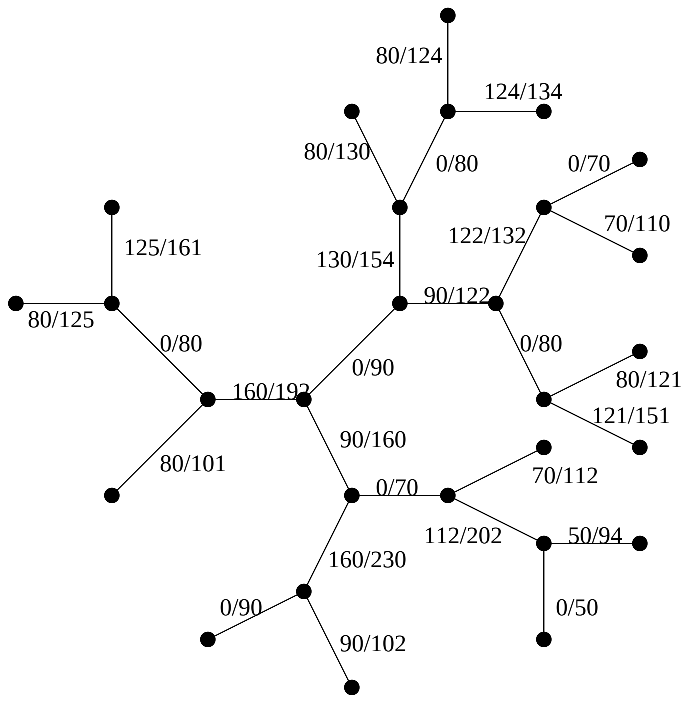
The propagating algorithm sets up a schedule with the passed initial edges being started at time zero. Time is propagated until at least one edge completes. A list is made of the completed edges, and they are removed from the graph. A list is made of the nodes involved in the completed edges. The nodes which are involved in a file transfer, and hence busy, at that time are also determined. For each node which just completed, the largest edge which can be started, that is which connects to a non-busy node, is started. This cycle of propagating time to the next time an edge stops continues.

This algorithm was applied to situation A and situation B. In both cases, it came up with an optimal solution, but one different than that presented in the body of this paper. For situation A, the graph looks like:



where the thin solid lines are started at time zero and finished at time one, the heavy dotted lines are started at time one and stop at time two, and the medium dotted lines are started at time two and end at time three. This is an optimal solution, as discussed in the body of this paper.

For situation B, a graph like the following results:



For this graph, the times have been quantized. This effectively means that all times are multiplied by ten from the data presented in the original problem. Each edge is labeled with two numbers separated by a slash. The first is the time which the transfer of that edge starts, and the second is the time which the transfer of that edge ends. This is also an optimal graph, finishing the last edge at time 230 (23.0 as presented in the body). However, it is different in both the edges it chooses to start with and the way in which it proceeds from there.