SCHEDULING TRANSMISSIONS IN A NETWORK

by

Alon Itai* and Michael Rodeh**

Technical Report #261

February 1983
Scheduling Transmissions in a Network

Alon Itai

Computer Science Department
Technion- IIT, Haifa, Israel

Michael Rodeh

IBM Israel Scientific Center
Technion City, Haifa, Israel.

ABSTRACT

Consider $k$ stations wishing to transmit messages over a network of channels to a common receiver. The capacity of a channel is the maximum amount of data which can be transmitted in a time unit. In addition to the transmission stations, the network contains switching nodes.

Given that the $j^{th}$ station has $\sigma_j$ messages ($j=1, \ldots, k$) to transmit, it is desired to find a schedule with minimum completion time $T$.

The amount of data sent over a channel may vary in time. A schedule is stationary if the amount of data sent in a time unit is constant throughout the schedule. It is first shown that for every schedule there exists a stationary schedule with the same completion time. Thus, the search for an optimum schedule is confined to stationary schedules.

The problem of finding an optimum stationary schedule is formulated as a multi-source single-sink network flow problem, in which the net amount of outgoing flow from each source (transmission station) within one time unit is $\sigma_j / T$. 
An $O(k|E||V|^2)$ time algorithm to find the minimum $T$ similar to Dinic's flow algorithm is suggested. Using Sleator and Tarjan's techniques an $O(k^2|E||V|\log |V|)$ algorithm is derived. The running time of both algorithms is independent of the $\sigma_i$'s and the capacities.
1. INTRODUCTION

Suppose that at $k$ stations there are data which have to be transmitted to a common receiver. The transmission should be conducted through a network consisting of memoryless nodes and finite-capacity channels. Even though the transmission is instantaneous the amount of data that can pass through a channel in a time unit is finite. Thus, it requires $m/c$ time units to transmit $m$ units of data over a channel of capacity $c$.

Since the nodes are memoryless, transmission is possible only if an entire path $\pi$ from the transmission station to the target node is available. Thus to complete the transmission of $\sigma_j$ units of data from station $j$, we could first secure a path $\pi_j$ from the station to the receiver, then transmit the $\sigma_j$ units of data within $\sigma_j/c(\pi_j)$ time units, where $c(\pi_j) = \min_{\pi \in \mathcal{P}_j} c(\pi)$. If all the stations are treated in this fashion one after the other, all the data is transmitted within $\sum_{r=1}^{k} \sigma_r/c(\pi_r)$ time units. However, at any instance several paths may share a channel, provided its capacity is not exceeded. Our scheduling problem is to find an optimum schedule, one which completes the transmission in minimum time. The problem was suggested to us by Prof. Adrian Segall.

The problem is modeled in Section 2. The amount of data sent over an edge may vary in time. A schedule is stationary if the amount of data sent in a time unit is constant throughout the schedule. In Section 3 it is first shown that for every schedule there exists a stationary schedule with the same completion time. Thus, the search for an optimum schedule is confined to stationary schedules.

The problem of finding a stationary solution can be formulated as a linear programming problem. However, to obtain an efficient algorithm Section 4 formulates a network flow problem. Section 5 develops a sufficient and necessary condition for a completion time to be minimum. Section 6 describes an
an optimal schedule, (where \( |V| \) is the number of nodes, both transmission stations and switching nodes, and \(|E|\) is the number of channels in the network). The solution is based on Dinic's network-flow algorithm [D]. An \( O(k^2|E||V|\log|V|) \) method, based on Sleator and Tarjan's implementation of Dinic's algorithm [ST], is presented in Section 7.

Finally, additional cost criteria are discussed:

2. THE FORMAL MODEL.

The network is modeled by a finite directed graph, \( G=(V,E) \), where \( V \), the set of vertices, corresponds to the transmission stations and switching nodes, and \( E \), the set of edges, corresponds to the channels. Each edge \( e \in E \) has a capacity \( c(e) > 0 \) - the maximum number of units of data which can be transmitted through the channel in a time unit. Let \( a_1, \ldots, a_k \in V \) be sources, corresponding to the transmission stations, and \( z \in V \) a sink - the common receiver. Let \( a_j \) be the amount of data to be transmitted from the source \( a_j \). The aim is to transmit all the data from the sources to the sink. To this end, a set \( I \) of instructions is found. Each instruction \( i \in I \) is a quadruple, \( i=(\text{start}_i,\text{finish}_i,n_i,q_i) \) where \( n_i \) is a directed path from some source \( a_j \) to the sink; \( q_i \) is the quantity to be transmitted along \( n_i \) starting at time \( \text{start}_i \) and finishing at time \( \text{finish}_i \) (>\(\text{start}_i\)). The load on an edge \( e \) at time \( t \) is

\[
load(e,t,I) = \sum_i q_i / (\text{finish}_i - \text{start}_i),
\]

where the summation is over all instructions \( i \in I \) whose path \( n_i \) contains \( e \) and \( \text{start}_i < t < \text{finish}_i \).

An instruction set \( I \) is feasible if the following constraints are fulfilled:

(i) The capacity constraint:

\[
\text{for all } e \in E: \quad 0 \leq load(e,t,I) \leq c(e);
\]
(ii) The evacuation constraint

$$\sigma_j = \sum_i q_i$$

where the summation is performed over all instructions $i \in I$ whose path $\pi_i$ starts at $a_j$.

Since the sources may be evacuated one after the other, there exists a feasible instruction set if and only if there is a directed path from every source $a_j$ to the sink $z$. This property of the graph can be checked in $O(|E|)$ time by conducting a breadth first search from the sink, scanning every edge in the reverse direction.

The completion time of an instruction set $I$ is the maximum completion time of all instructions $i \in I$, i.e. $\max_{i \in I} finish_i$.

An instruction set is optimum if its completion time, $T$, is minimum; i.e., it is equal to $\min_I \max_{i \in I} finish_i$ where the minimization is over all feasible instruction sets $I$.

3. STATIONARY INSTRUCTION SETS

An instruction set $I$ is stationary if all the instructions $i \in I$ start and finish at the same time, i.e.

$$\text{for all } i \in I \quad \text{start}_i = 0 \text{ and } \text{finish}_i = T.$$  

The following theorem reduces the problem to stationary instruction sets.

Theorem 1: Let $I$ be an instruction set with completion time $T$, then there exists a stationary feasible instruction set $S$ with the same completion time.

Proof: The instruction set $S$ is defined to be the average over time of $I$. Formally, for every instruction $i = (\text{start}_i, \text{finish}_i, \pi_i, q_i)$, let $s_i = (0, T, \pi_i, q_i)$. The set $S$ consists of all the instructions $s_i$. 

Technion - Computer Science Department - Technical Report CS0261 - 1983
By the definition, $S$ is stationary. To show feasibility, we note that the evacuation constraint holds since it holds for every instruction $i \in I$ with the same $q_i$ and the same path $\pi_i$.

It remains to show that the capacity constraint holds for $S$. By definition of $S$

$$\text{for all } e \in E \text{ and for all } 0 < t \leq T: \text{load}(e, t, S) = \sum_{i \in I} \sum_{e \in \pi_i} q_i / T.$$  \hspace{1cm} (1)

The capacity constraint for $I$ is $\text{load}(e, t, I) \leq c(e)$. Consequently,

$$\int_0^T \text{load}(e, t, I) dt \leq c(e) T.$$  \hspace{1cm} (2)

However,

$$\int_0^T \text{load}(e, t, I) dt = \sum_{i \in I} \sum_{e \in \pi_i} q_i.$$  \hspace{1cm} (3)

Combining (1), (2), and (3) yields,

$$\text{load}(e, t, S) = \frac{1}{T} \sum_{i \in I} \sum_{e \in \pi_i} q_i \leq c(e).$$

**Corollary 1:** If there exists an optimum feasible instruction set then there exists a stationary optimum feasible instruction set.

4. A NETWORK FLOW SOLUTION

To find an optimum stationary feasible instruction set a network flow problem $[FF, D, E]$ is defined. The flow on an edge is the amount of data transmitted through it in one time unit. Therefore, the flow $f(e)$ on the edge $e$ must fulfill the capacity constraint:

$$\text{for all } e \in E: 0 \leq f(e) \leq c(e).$$

The flow must be conserved at the vertices:
for all \( v \in V - \{ a_1, \ldots, a_k, z \} \): \[ \sum_{(u,v) \in E} f(u,v) - \sum_{(v,w) \in E} f(v,w) = 0. \]

Since every source must be evacuated, the following is imposed:

for all \( 1 \leq j \leq k \) \[ \sum_{(a_j,u) \in E} f(a_j,u) - \sum_{(w,a_j) \in E} f(w,a_j) = \sigma_j / T. \]

We wish to minimize \( T \).

We claim that given a flow which satisfies these constraints for some value \( T \), there exists a stationary feasible instruction set with the same completion time.

To find the instruction set \( I \) repeat the following procedure until all the \( \sigma_j \)'s are equal to zero. Perform a depth first search from the sink backwards along the edges with positive flow, until either a circuit is closed or a source \( a_j \) is reached. In the former case, find a bottleneck, an edge \( b \) with minimum flow on the circuit, and decrease the flow by \( f(b) \) units along the circuit. The edge \( b \) no longer has positive flow. In the latter case, a path \( \pi \) from some source, say, \( a_j \) has been found. Let \( b \) be a bottleneck, an edge with minimum flow on \( \pi \), and let \( q = \min(\sigma_j, f(b)) \). Issue the instruction \( i = (0, T, \pi, q) \), decrease the flow along \( \pi \) by \( q \) units and decrease \( \sigma_j \) by \( q \).

At each iteration either the flow on some edge (the bottleneck) or one of the \( \sigma_j \)'s has become zero. Therefore, there may be at most \( k + |E| = O(|E|) \) iterations.

The following result is also implied:

**Corollary 2:** A transmission scheduling problem which has an instruction set with completion time \( T \) has an instruction set with the same completion time and no more than \( k + |E| \) instructions.

Since at most one circuit is closed at every iteration, each iteration requires \( O(|V|) \) time. Thus, the following result is obtained:
Lemma 2: Given a flow, the corresponding instruction set may be constructed within $O(|V| \cdot |E|)$ time.

Remark 1: There are flow networks the maximal flow of which can be achieved only by $\Phi(|E|)$ augmenting paths, each of length $\Phi(|V|)$. Thus, in the worst case, $\Phi(|V| \cdot |E|)$ is also a lower bound on the time required to find the augmenting paths, and an optimal stationary schedule.

Remark 2: The problem of finding an optimal stationary instruction set with the minimum number of instructions is NP-hard, since finding the minimum number of augmenting paths to achieve a maximal flow is also NP-hard. (Exercise 10.20 [AHU] suggested by Prof. S. Even.)

5. PROPORTIONAL FLOW

In the previous section the transmission scheduling problem was formulated in terms of time independent flow. The formulation is different than the standard network flow problems in the appearance of $T$ in the evacuation constraints and its minimization:

$$1 \leq j \leq k: \sum_{(a_j,u) \in E} f(a_j,u) - \sum_{(w,a_j) \in E} \text{flow}(w,a_j) = \sigma_j / T.$$  

If the network contains paths from every source to the sink then an initial value of $T$ and the corresponding flow function can easily be found (i.e., $T = \sum_{j=1}^{k} \sigma_j / c(\pi_j)$, where $\pi_j$ is a path from $a_j$ to $z$ and $c(\pi_j)$ is the minimum capacity of its edges). The minimum value of $T$ for which there exists a flow function could be found by binary search, i.e., in a logarithmic number of iterations, depending on the required precision. The running time of this method depends on the
values of the capacities and the $\sigma_j$'s.

In the remainder of this paper, we propose a method for finding the minimum $T$ which is independent of the $\sigma_j$'s and the capacities. The method successively reduces $T$ by increasing the flow in the network. To this end, a set of augmenting paths leading from the sources to the sink is constructed using an auxiliary graph.

Let $f$ be a flow function in $G$. Define the network $G' = (V, E')$ where

$$E(G') = \{(u, v); f(u, v) < c(u, v) \text{ or } f(v, u) > 0\}.$$  
and the capacities of $G'$ (called residual capacities) are defined

$$r(u, v) = c(u, v) - f(u, v) + f(v, u).$$

Even if the network $G$ is undirected, $G'$ is directed.

Augmenting paths in $G$ from the sources to the sink exist if and only if there exist directed paths from the sources to the sink in $G'$. Assume that for all $1 \leq j \leq k$ a path $\pi_j$ from $a_j$ to $z$ in $G'$ has been found. Let

$$\varepsilon = \min\{r(e)/\sum_j \sigma_j\}$$

where the minimization is over all the edges which participate in some $\pi_j$ and for each edge $e$ the summation is over all paths $\pi_j$ which pass through $e$. Pushing $\varepsilon \sigma_j$ through $\pi_j$ for every $j$ is a proportional augmentation. After a proportional augmentation all the capacity constraints hold and either some edge of $G$ has become saturated (its residual capacity becomes zero) or the flow of some edge becomes zero. The resultant flow corresponds to a smaller completion time. It is possible to proceed with proportional augmentations until at least one of the sources, say $a_j$, becomes disconnected from the sink, i.e. there exists no augmenting path from $a_j$ to $z$. 

Technion - Computer Science Department - Technical Report CS0261 - 1983
Lemma 3: If for a given flow function no proportional augmentation is possible then the corresponding value of $T$ is minimum.

Proof: Let $f$ be a flow function and $T$ its corresponding completion time. Assume that $T$ is not minimum but still no proportional augmentation exists. Let $I'$ be an instruction set whose completion time $T'$ is strictly less than $T$. From Theorem 1 we may assume that $I'$ is stationary. Let every instruction $i' \in I'$ be of the form $(0, T', \pi_i', q_i')$. It is obvious that $q_i'/T'$ units of flow may be pushed along $\pi_i'$ without violating the capacity constraint and the conservation law but still transmitting all the data from the sources. By pushing only $q_i'/T$ units of flow through $\pi_i'$, for every $i' \in I'$, a flow function $g$ is obtained. With respect to $g$, no residual capacity is zero. Therefore, there exists an augmenting path from every source $a_j$ to $z$ ($1 \leq j \leq k$).

Since there is no proportional augmentation for the flow function $f$, then some source, say $a_1$, is disconnected from $z$ with respect to $f$. Let $G^*=(V, E^*)$ with capacities $c^*$, a single source $a_1$ and sink $z$ be the following network:

(i) For all $e \in E$: $e \in E^*$ and $c^*(e)=c(e)$.

(ii) For all $2 \leq j \leq k$: $(a_1, a_j) \in E^*$ and $c^*(a_1, a_j)=q_j/T$.

By pushing $q_j/T$ units of flow through the edge $(a_1, a_j)$ ($j=2, \ldots, k$), both $f$ and $g$ can be turned into a flow in $G^*$ with a flow value (the total flow leaving the source) equal to $\frac{1}{T} \sum_{j=1}^{k} q_j$. The flow $g$ is not maximum, since there exists an augmenting path from $a_1$ to $z$. Therefore $f$ is not maximum either. An augmenting path leading from $a_1$ to $z$ with respect to $f$ cannot use any new edge of the type $(a_1, a_j)$ because all these edges are saturated. Thus, the graph $G$ itself has an augmenting path from $a_1$ to $z$ with respect to $f$, a contradiction.
Lemma 3 constitutes a necessary and sufficient condition for a flow completion time $T$ to be minimum.

6. A POLYNOMIAL ALGORITHM

6.1. Description of the Algorithm

The previous algorithm allowed complete freedom in choosing the augmenting paths. In fact it is analogous to Ford and Fulkerson's network flow algorithm [FF]. By using a refinement analogous to that of Dinic [D] an $O(k |V|^2 |E|)$ algorithm is obtained.

The algorithm proceeds in phases. In each phase a new graph (the layer graph) is constructed and is used to find augmenting paths. The notion of the layer graph $L_i$ is explained first. Let $f_i$ be the flow at the end of the $i-1$st phase. Conduct a breadth first search in $G^{f_i}$ (the auxiliary graph defined in the previous section) from $z$ traversing the'edges backwards. Let the level of a vertex $v$ (denoted by $\text{level}_i(v)$) be the length of the shortest directed path from $v$ to $z$. (If there is no path from the vertex $v$ to $z$ then $\text{level}_i(v) = \infty$.) Let $L_i = (V, E_i)$ be the following network:

(i) \[ E_i = \{(u,v) \mid u,v \in V, (u,v) \in G^{f_i} \text{ and } \text{level}_i(u)=\text{level}_i(v)+1\}. \]

(ii) \[ c_i(u,v) = c(u,v) \text{ for all } (u,v) \in E_i. \]

(iii) The sources are $a_j, 1 \leq j \leq k$.

(iv) The sink is $z$.

Start with $f(e) = 0$ for all $e \in E$, and $T = \infty$. If for any source $a_j$, $\text{level}_i(a_j) = \infty$ then there exists no augmenting path from $a_j$ to $z$. Therefore, there exists no proportional augmentation. In such a case the algorithm terminates.
For every source \(a_j\) a path \(\pi_j\) is constructed from \(a_j\) to \(s\) in \(L_4\) (Section 6.3 will explain how to find such paths). The flow is augmented as explained in the previous section. Every proportional augmentation causes at least one edge in \(L_4\) to become saturated and therefore may be deleted. This process continues until some source becomes disconnected from the sink; whereupon the phase terminates.

### 6.2. The Number of Phases

The following lemma is the key to the efficiency of the algorithm:

**Lemma 4:** For all \((u,v) \in G^{l+1}\), \(\text{level}_i(u) - 1 \leq \text{level}_i(v)\).

**Proof:** If \(\text{level}_i(u) - 1 > \text{level}_i(v)\), the difference in levels between \(u\) and \(v\) is at least two, hence \((v,u),(u,v) \in L_4\) and the residual capacity of \((u,v)\) with respect to \(f_{i-1}\) is zero. Since \((v,u),(u,v) \in L_4\) the flow through these edges did not change during the \(i\)th phase and the residual capacity of \((u,v)\) is also zero with respect to \(f_i\). Therefore, \((u,v) \in L_{i+1}\), a contradiction.

**Lemma 5:** For all vertices \(u\), \(\text{level}_i(u) \leq \text{level}_{i+1}(u)\).

**Proof:** Let \(\pi=(u=v_0, \ldots, v_m=s)\) be a path in \(L_{i+1}\). Define \(\lambda=(\lambda_0, \ldots, \lambda_m)\) such that \(\lambda_h = \text{level}_i(v_h)\). Thus \(\lambda_m=0\). By Lemma 4, \(\lambda_h \leq \lambda_{h+1}+1\). Therefore, \(\lambda_0 \leq m\). Substituting for \(\lambda_0\) and \(m\) yields \(\text{level}_i(u) \leq \text{level}_{i+1}(u)\).

**Lemma 6:** For every \(i\) there exists at least one source \(a_p\) (\(p=p(i)\)) for which \(\text{level}_i(a_p) \leq \text{level}_{i+1}(a_p)\).

**Proof:** Repeat the proof of the previous lemma with \(u=a_p\). If \(\lambda_h = \lambda_{h+1}+1\) for \(h=0,\ldots,n-1\) then \(\pi\) is a path of \(L_4\), contrary to the fact that \(a_p\) became
Disconnected in \( I_k \). Therefore, for some \( 0 \leq h < m \), \( \lambda_h < \lambda_{h+1} + 1 \). Thus \( \lambda_0 < m \) and 
\( \text{level}_i(a_p) < \text{level}_{i+1}(a_p) \).

**Corollary 3:** The number of phases is bounded by \( k |V| \).

**Proof:** In each phase one of the sources moves further away from the sink and no source gets nearer to the sink. Since the distances are bounded by \( |V| \), the number of phases is bounded by \( k |V| \).

### 6.3. Finding Paths in the Layer Graph \( I_k \)

To obtain an efficient algorithm, a method for finding the augmenting paths must be specified. We shall use a method similar to that of Dinic for finding a blocking flow. (A more efficient method will be described in Section 7.) The length of each augmenting path may be of the order of \( |V| \). There are \( k \) such paths and consequently, the sum of their lengths may be as large as \( O(k |V|) \). Therefore, the naive approach leads to an \( O(k |V|) \) algorithm for finding a proportional augmentation. To reduce computation time, observe that once two augmenting paths meet, the continuation of one of them may be deleted since the other one may be used to reach \( z \). This consideration leads to a tree structure of the augmenting paths, called *augmenting tree*. Note that if an augmenting tree is decomposed back into augmenting paths, the sum of their lengths may still be \( O(k |V|) \).

To keep the computation time low, the entire tree should be processed rather than each augmenting path separately.

To construct an augmenting tree, start with the tree \( \{z, \emptyset \} \), i.e. a tree which contains the vertex \( z \) and no edges. For every source \( a_j \) conduct a depth-first-search from \( a_j \) on the current layer graph \( I_k \). The search continues until a vertex belonging to the current tree is reached. In the course of the
search some edges \((u,v)\) may be encountered such that no edge emanates from \(v\). Such edges are called dead ends. They are deleted from \(L_4\) since they cannot participate in any augmenting path of \(L_4\). As a result of this deletion the edge through which \(u\) was entered may also become a dead end, and it also is deleted. This process continues until a tree-vertex is reached. Every edge scanned is either a dead end or part of the augmenting tree. Since there are at most \(|E|\) edges in \(L_4\), the processing time of dead ends in an entire phase is \(O(|E|)\). Since there are at most \(|V| - 1\) edges in the augmenting tree the time spent on its construction (not counting dead ends) is \(O(|V|)\):

Each time a tree is found, a proportional augmentation (to be discussed below) takes place and causes at least one edge to become saturated and be deleted from \(L_4\). Therefore, at most \(O(|E|)\) augmenting trees are found, and the time required to find all of them is at most \(O(|V| |E|)\) per phase.

6.4. Proportional Augmentation and Updating

In order to perform proportional augmentation it is necessary to compute the value of \(z\) first (see Section 5). To this end, for every edge \(e\) of the augmenting tree, we have to compute the values of \(q(e) = \sum \sigma_j\), where the sum is over all sources \(a_j\) for which the path \(\pi_j\) passes through \(e\). This is easy to do since \(\dot{q}(v,w) = \sum q(u,v)\) for all edges \((u,v)\) which belong to some augmenting path entering \(v\). If \(v\) is a source \((v = a_j)\) then \(\sigma_j\) should be added. By scanning the tree from the leaves to the root \(z\), \(O(|V|)\) time suffices to compute \(q\) for all the edges of the augmenting tree.

The remaining updating operations require \(O(|V|)\) time per augmenting tree. Since there may be at most \(|E|\) augmenting trees per phase, the entire updating requires \(O(|V||E|)\) time per phase.
6.5. The Time Bound

Theorem 7: An optimum stationary feasible instruction set for a given solvable transmission scheduling problem with \( k \) sources can be found in \( O(k |V|^2 |E|) \) time.

Proof: To find a flow function which corresponds to the minimum completion time the above algorithm proceeds in phases. Each phase consists of:

(i) Finding the layer graph \( L_4 \) \( (O(|E|) \) time);

(ii) Finding augmenting trees in \( L_4 \) \( (O(|V| |E|) \) time);

(iii) Updating the edges of the augmenting trees \( (O(|V| |E|) \) time).

Since there may be at most \( k |V| \) phases, the entire algorithm requires \( O(k |V|^2 |E|) \) time.

Given a flow function, constructing a corresponding instruction set takes \( O(|V| |E|) \) time (Lemma 2).

7. A FASTER ALGORITHM

In Section 6 we used Dinic's algorithm to find a blocking flow of the layer graph in \( O(|V| |E|) \) time. Karzanov [K] and Malhotra et al. [MPM] do it in \( O(|V|^2) \) time by finding sets of augmenting paths rather than each path individually. Since our flow problem requires us to find a bottleneck of an augmenting tree their algorithms are not immediately applicable. However, the newer algorithms of Galiil and Naamad [GN], Shiloach [S] and Sleator and Tarjan [ST] find paths individually using clever data structures to reduce the time for finding and updating each path, thus they are more suitable to our purposes. Since the Sleator-Tarjan algorithm is more efficient we shall use it. Let us start by describing their algorithm.
7.1. The Sleator-Tarjan Algorithm

The algorithm provides a data structure for maintaining a vertex-disjoint forest of edge-weighted rooted trees called *dynamic trees*. The operations defined on these trees are:

(a) \( \text{link}(v, w, p) \): Add an edge of cost \( p \) from \( v \) to \( w \), \( v \) is a root of a tree, and \( w \) does not belong to that tree, thereby adding the dynamic tree rooted at \( v \) to the tree to which \( w \) belongs.

(b) \( \text{cut}(v, w) \): Delete the edge \( (v, w) \), thus creating two dynamic trees instead of one.

(c) \( \text{root}(v) \): Find the root of the dynamic tree containing the vertex \( v \).

(d) \( \text{cost}(v, w) \): Find the cost of the edge \( (v, w) \).

(e) \( \text{find\_min}(v) \): Return a minimum cost edge belonging to the path from \( v \) to \( \text{root}(v) \).

(f) \( \text{update}(v, p) \): Add the real number \( p \) to the costs of all the edges on the path from \( v \) to \( \text{root}(v) \).

(g) \( \text{sons}(v) \): A pointer to a list of all the sons of \( v \) in the dynamic tree containing \( v \).

(h) \( \text{father}(v) \): Find the unique father vertex of \( v \) in the dynamic tree.

The data structure requires \( O(|V| + m \log |V|) \) time to carry out any sequence of \( m \) of the above "Sleator-Tarjan" operations.

The following operation is not directly supported by the data structure but can be implemented using a constant number of the previous operations, as will be explained in Section 7.8.

(i) \( \text{dca}(v, w) \): Deepest common ancestor of \( v \) and \( w \), i.e. the vertex furthest away from the root which is a dynamic tree ancestor of both \( v \) and \( w \).
Remark: To simplify the presentation a new vertex $z'$ is added to the network to serve as the sink and the old sink $z$ becomes an ordinary vertex connected to $z'$ by an edge of infinite capacity.

7.2. Our algorithm - an overview

In this section we prove the following theorem:

**Theorem 8:** An optimum stationary feasible instruction set for a given solvable transmission scheduling problem with $k$ sources can be found in $O(k^2|V||E|\log|V|)$ time.

To prove the theorem we present an algorithm which uses a data structure with a large number of operations, some of which are exactly those of Sleator and Tarjan, and for the others we show an implementation which uses at most $k$ Sleator-Tarjan operations. It will be shown that the algorithm requires a total of $O(|E|)$ such operations per phase. Thus each phase costs at most $O(k|E|\log|V|)$. The complexity claimed in Theorem 8 follows since there are at most $O(k|V|)$ phases. The remainder of this section describes the algorithm and the required operations.

7.3. Our algorithm - a detailed description

In the proposed algorithm an augmenting tree is manipulated rather than augmenting paths in standard network flow algorithms. The augmenting tree $A$ is a subgraph of the layer graph, every leaf is a source, and the root is the sink $z'$. In Figure 1 the augmenting tree edges appear as solid lines and other edges as broken lines. Even though an augmenting tree may have up to $|V|$ vertices, there may be at most $k-1$ vertices of indegree greater than 1. A vertex of an augmenting tree is a *junction* if it is either the sink $z'$, a source or a vertex of indegree greater than 1. Let $J = \{j_1, j_2, \ldots, j_m\}$ be the set of junctions. There are at
most $2k$ of them. The junctions define a set $B = \{B_1, B_2, \ldots, B_{m-1}\}$ of branches where a branch is a path in the augmenting tree connecting two neighboring junctions. Thus, in Figure 1

$$X = \{a_1, \ldots, a_n, e', u_A\},$$

$$B = \{(e_1, e_2), (e_3, e_4), (e_5), (e_7, e_8), (e_9, e_{10})\}.$$ 

To explain the details of the algorithm, let us review the main steps of the proportional flow algorithm. The algorithm first finds an initial augmenting tree and then repeats the following steps until one of the sources becomes disconnected from the sink:

Step 1. Compute $q(v)$ for every augmenting tree edge $v$. Then find a bottleneck

$$b_A = (u, v)$$

- an edge $v$ on which $\frac{r(v)}{q(v)}$ attains its minimum.

Step 2. Push $\frac{r(b_A)}{q(b_A)}$ units of flow from every source $a_j$ through $A$ to $e'$. 

Step 3. Delete $b_A$ from the layer graph.

Step 4. Perform a depth-first-search from $u$ until encountering a vertex of the augmenting tree, thus reconnecting the disconnected sources to the sink.

Just as in Sleator-Tarjan's algorithm, the costs are the residual capacities and pushing flow is implemented by modifying these capacities. When the residual capacity reaches zero the edge is deleted from the layer graph.

To implement our algorithm we let Sleator-Tarjan's data structure maintain a forest of dynamic trees one of which contains the augmenting tree as a subtree. To facilitate controlling the structure of $A$ we also maintain the skeleton $S_A$ of $A$, a tree with junctions as vertices and branches as edges and support the operation $\text{find\_branch}(w)$ defined by:
(1) \textit{find\_branch}(w): Find the unique branch to which \(w\) belongs and return \texttt{nil} if \(w\) does not belong to the augmenting tree. The implementation of this operation is explained in Section 7.5.

Notice that we maintain three different trees:

(1) The augmenting tree, which is not maintained explicitly.

(2) A dynamic tree which contains the augmenting tree and is maintained by Sleator-Tarjan’s data structure.

(3) The skeleton, which is manipulated directly by our algorithm. Figure 1 shows the skeleton of the augmenting tree of Figure 1.

We now discuss each step of the algorithm separately:

Step 1: First \(q(e)\) must be computed for every augmenting tree edge \(e\). A straightforward observation is that the value of \(q\) changes only at the junctions and is fixed along each branch. Let \(C = (c_1, c_2)\) be an edge of the skeleton \(S_A\) starting at the junction \(c_1\) and ending at the junction \(c_2\) and let \(q(C)\) denote the value of \(q\) on every augmenting tree edge on the path from \(c_1\) to \(c_2\).

To compute \(q(C)\) for every branch \(C = (c_1, c_2)\), notice that

\[
q(C) = \sum_j \sigma_j,
\]

where the summation is over all sources \(a_j\) belonging to the subtree rooted at \(c_1\). Thus computing \(q(C)\) for all branches \(C\) may be done by a single postorder traversal of \(S_A\), requiring \(O(k)\) operations.

Now that \(q(C)\) is known for every branch \(C\), \(\min_{e} \frac{r(e)}{q(C)}\) can be found by computing \(\min_C \min_{e \in C} \frac{r(e)}{q(C)}\). To compute \(\min_C \min_{e \in C} \frac{r(e)}{q(C)}\) the following operations are performed:

(i) Let \(f := \text{father}(c_2)\) and \(d := \text{cost}(c_2, f)\).
(ii) $\text{cut} (c_2,f)$.

(iii) $\min_{e \in \text{q}(e)} \frac{r(e)}{q(C)} := \frac{\text{find}_{\text{min}}(c_1)}{q(C)}$.

(iv) $\text{link} (c_2,f,d)$.

To find $\min_{e \in \text{q}(e)} \frac{r(e)}{q(C)}$ we perform the above procedure for every branch $C$. The branches are the edges of the skeleton $S_d$. Notice that overall there are $O(k)$ operations of types (a), (b), (d), (e) and (h).

**Step 2:** The amount $g$ of flow pushed through every augmenting tree edge of a branch $C = (c_1,c_2)$ is the same. Therefore the mechanism of pushing this flow is similar to that of computing $q(C)$: Finding $g$ is done by a single postorder traversal of $S_d$. Updating the capacities is done as follows:

(i) Let $f := \text{father}(c_2)$ and $d := \text{cost}(c_2,f)$.

(ii) $\text{cut}(c_2,d)$

(iii) $\text{update}(c_1,-g)$

(iv) $\text{link}(c_2,f,d)$.

We see that $O(k)$ operations of type (a), (b), (d) and (h) suffice.

**Steps 3 and 4:** During Step 1 a bottleneck $b=(b_1,b_2)$ and a branch $C=(c_1,c_2)$ containing $b$ are found (a dynamic tree is depicted in Figure 2.1 and its skeleton in Figure 2.1'). When $(b_1,b_2)$ is removed from the augmenting tree (Figure 2.2) the branch $(c_1,c_2)$ to which it belongs should be removed from the skeleton $S_d$. If as a result of this deletion $c_2$ ceases to be a junction (its indegree becomes equal to 1 and it is neither a source nor a) then $c_2$ should be deleted from $S_d$ and its unique son should be directly connected to $c_2$'s father (Figure 2.2'). Note that the skeleton has been dissected to two parts, one rooted at $z'$ and the other rooted at $c_1$. 

Technion - Computer Science Department - Technical Report CS0261 - 1983
To complete the implementation of steps 3-4 we have to reconnect $b_1$ to the augmenting tree. A depth first search is initiated at $b_1$. Let $u$ be the vertex most recently visited by the search. If $u$ is a dead-end in the layer graph then we backtrack from $u$ and the search continues from its son $w$. (The case in which $u$ has more than one dynamic tree son will be discussed in Section 7.4.) If $u$ is one of the sources the current phase of the algorithm terminates.

When the depth first search enters a vertex $v$ we perform $\text{link}(u,v,\text{cost}(u,v))$. If $v$ does not belong to the dynamic tree, the search continues from $v$. If $v$ belongs to the dynamic tree the search is finished and we turn to updating the skeleton. (A special bit associated with each vertex may denote whether a vertex belongs to the tree.) Two cases may arise:

**Case 1:** $v$ is a junction. Connect $c_1$ to $v$ in $S_4$ (Figures 2.3, 2.3').

**Case 2:** $v$ is not a junction. Perform $C' := \text{find\_branch}(v)$. Two sub-cases arise:

**Sub-case 2.1:** $v$ belongs to the augmenting tree. To update the skeleton the branch $C' = (c_1',c_2')$ is replaced by $(c_1',v)$ and $(v,c_2')$, i.e., $v$ becomes a new junction. Also, the branch $(c_1,v)$ is added thereby reconnecting the skeleton (Figures 2.4, 2.4').

**Sub-case 2.2:** $v$ does not belong to the augmenting tree. (This is possible if after $v$ had entered the dynamic tree (and the augmenting tree) there was a bottleneck in the branch containing it causing $v$ to be deleted from the augmenting tree but not from the dynamic tree, see for example the edge $(b_2,c_2)$ in Figure 2.3). Perform $y_z := dca(v,x)$ for all junctions $x$. Let $y$ be the $y_z$ furthest away from the root; $y$ belongs both to the augmenting tree and to the path from $v$ to the root (Figure 2.5). If $y$ is a junction add $(c_1,y)$ to $S_4$ (Figure 2.5'). Otherwise, let $(c_1',c_2') := \text{find\_branch}(y)$ (Figure 2.6). To update the skeleton the branch $C' = (c_1',c_2')$ is replaced by $(c_1',y)$ and $(y,c_2')$, i.e., $y$ becomes a new junction. Also, the branch $(c_1,y)$ is added thus reconnecting the skeleton (Figure 2.6').
7.4. Multiple Backtracking

In step 3 we noticed that the vertex \( v \) from which we backtrack might have more than one dynamic tree son. Let \( \text{sons}(v) = \{w_1, \ldots, w_i\} \). All the edges \((w_i, u)\) can no longer be part of the augmenting tree and should be deleted by a \( \text{cut}(w_i, u) \) operation. However this dissects the dynamic tree into \( i+1 \) trees (whereas in Section 7.3 we had at most 2 trees). In general, we have a forest of dynamic trees, some of which contain sources. For each source \( a_j \) let \( \tau_j = \text{root}(a_j) \). If \( \tau_j \neq u \), then perform a depth-first-search from \( \tau_j \), until either we backtrack from \( a_j \) in which case this phase of the algorithm terminates, or we reach a vertex \( v \) of some dynamic tree \( D_i \). If \( v' \) is the last vertex before \( v \), the tree to which \( a_j \) belongs is added to \( D_i \) by executing \( \text{link}(v', v, \text{cost}(v', v)) \). The corresponding skeletons need also be updated in a fashion similar to Section 7.3.

7.5. Implementation of \( \text{find\_branch}(w) \)

\( \text{find\_branch}(w) \) yields an edge \((d_1, d_2)\) of the skeleton. To find \( d_2 \) we perform the following for all junctions \( x \neq \text{root}(w) \):

(i) \( \text{Let } f := \text{father}(x), x := \text{root}(w) \) and \( p := \text{cost}(x, f) \).
(ii) \( \text{cut}(x, f) \).
(iii) \( d_x := \text{root}(w) \).
(iv) \( \text{link}(x, f, p) \).

If \( x \) belongs to the dynamic tree path from \( w \) to \( r \) then \( d_x = z \), the vertex \( d_2 \) belongs to this path and is the junction furthest away from \( r \). If for all junctions \( x \neq d_x \) then \( d_x \) is \( r \).

If \( w \) belongs to the augmenting tree then it belongs to a branch \((d_1, d_2)\) for some \( d_1 \) where \( d_1 \) is the only skeleton son \( z' \) of \( d_2 \) for which \( w = \text{root}(z') \) when the augmenting tree-edge emanating from \( w \) is temporarily cut. If for all sons \( z' \) of \( d_2 \) \( \text{root}(z') = r \) then \( w \) does not belong to the augmenting tree, i.e.
\textit{find\_branch}(w) = \text{nil}.

7.6. Implementation of \textit{dca}(v,w)

To implement \textit{dca}(v,w) we apply the Sleator-Tarjan algorithm to the layer graph with a different cost function. Let \( n = |V| - 1 \). Assign a cost of \( n - i \) to an edge at distance \( i \) from \( z' \). Thus the costs are between 0 and \( n-1 \). To find \textit{dca}(v,w) perform the following steps:

\begin{verbatim}
update(v, n);
update(w, n);
(u1, u2) = find_min(v);
update(v, n);
update(w, n);
return(u)
\end{verbatim}

As a result of the first two \textit{update}'s, the edges common to the \((v, z')\) path and the \((w, z')\) path decreased by \(2n\), whereas all the other edges decreased by at most \(n\). Thus only the common edges have cost less than \(-n\) and among these the one furthest away from \(z'\) has the lowest cost. Thus \(u\) is the deepest common ancestor.

Note that the above procedure does not work properly if \(z' = \textit{dca}(v,w)\). However, in our algorithm \(z'\) is not a result of any \textit{dca} operation. Also, neither the dynamic tree nor these special costs change as a result of these operations.
8. CONCLUSIONS

The transmission scheduling problem was shown to have a stationary solution. Based on this fact, the problem was reformulated as a flow problem with parametrized capacities and solved by an extension of standard network flow algorithms. An interesting observation is that if there exists only a single source, the algorithm is basically the same as Sleator and Tarjan's.

We may assign costs to the edges, thus it costs $\gamma(e)$ to transmit $q$ messages on an edge $e$ of cost $\gamma(e)$. A minimum cost solution may be found as follows: First find an optimum stationary solution disregarding the costs. If its completion time is $T$ construct a flow network consisting of the original network, a new source $a$ and edges $(a,a_j)$ $j=1, \ldots, k$ with capacities $\sigma_j/T$, and zero cost. The old sources become ordinary vertices. Now apply Edmonds and Karp's [EK] minimum-cost max-flow algorithm to the new network (starting with zero flow). The total flow is $(\sum \sigma_j)/T$, and the new edges are all saturated. This flow induces a minimum-cost flow in the original network. The corresponding minimum cost instruction set can be found as in Section 4.
REFERENCES


Figure 1

Figure 1'
Figure 2.1
Figure 2.2

Figure 2.2'
Figure 2.3

Figure 2.3'
Figure 2.4
Figure 2.5

Figure 2.5'
Figure 2.6