This work was partially funded by the National Science Foundation under grant numbers MCS-81/05894, and by the IBM Corporation and the State of California through project MICRO.

FAST ALGORITHMS FOR BIPARTITE NETWORK FLOW

Dan Gusfield[1]
Charles Martel[2]
David Fernandez-Baca[2]

YALEU/DCS/TR-356

January, 1985

[1]Department of Computer Science, Yale University

[2]Department of Computer Science, University of California, Davis

Table of Contents

Abstract

We discuss network flow in a bipartite graph G with node sets S and T. We show that Karzanov's algorithm runs in time $O(|S|^2|T|)$ on G, and that the MPM algorithm can be modified to run in the same bound. For the common case that the degree of each node in T is bounded by a constant, we show that a modified version of the MPM algorithm runs in $O(|S|^3 + |S||T|)$ on G. The importance of these results comes from the common occurrence of problems that can be modeled and solved as network flow on bipartite graphs where $|S| << |T|$. In many applications involving bounded degree of the nodes in T, $|T| = \Theta(|S|^2)$, and our results reduce the best bounds on bipartite flow from $O(|S|^6)$ to $O(|S|^3)$. We use our results on bipartite flow to obtain the fastest know bounds on several combinatorial problems, and mention nine additional problems involving bipartite flow where $|S|$ is often much smaller than $|T|$.

1. Introduction

The maximum flow and minimum cut problem is used both as an algorithmic and mathematical tool to model and solve hundreds of interesting and practical problems in combinatorics and combinatorial optimization, and the importance of the network flow problem is well established. A large subset of these network flow problems are modeled and solved on graphs which are *bipartite*, and which have the property that the set of nodes, S, on one side of the graph is much smaller than the set of nodes, T, on the other side of the graph; we list and discuss several such problems in section four.

Main Results

In this paper we show that the flow problem (finding the maximum flow and the minimum cut) on such bipartite graphs can be solved far more efficiently than has been previously established. In particular, we show that the bipartite flow problem can be solved in time $O(|S|^2|T|)$ when $|S| \leq |T|$, and for the class of graphs where the degree of each node in T is bounded by a constant, bipartite network flow can be solved in time $O(|S|^3+|S||T|)$. Many network flow problems of importance have exactly this bounded degree structure, as we will later illustrate. We then express the main result in terms of the size of the maximum independent set in the graph, and show that our results imply faster network flow algorithms for graphs with one large independent set of nodes; the bipartite graphs are a special case of this class. We also mention simpler, slower network flow algorithms which can be shown to run faster on bipartite graphs than the general bounds suggest.

Presently known general upper bounds for network flow imply $O((|S|+|T|)^3)$ or $O((|S|+|T|)|E|\log|E|)$ time bounds for bipartite flow. For most of the applications we consider, $|T| = \Theta(|S|^2)$, and so our results reduce the best bounds from from $O(|S|^6)$ to $O(|S|^4)$ and often to $O(|S|^3)$. For many of the problems discussed, there are known methods for solving the problems which do not use bipartite flow, and which previously looked superior to solution methods based on bipartite flow. For each problem discussed in this paper, our bounds on bipartite flow yield methods that beat or equal the best previously known bounds of any of the other solution methods.

The better running times presented in this paper are achieved by a closer than standard analysis of well known flow algorithms, and by simple modifications of these algorithms: we use a new charging argument to count the work in the well-known MPM and Karzanov algorithms; we establish the $O(|S|^2|T|)$ bound in MPM by the use of F-heaps in the implementation, but it is the the charging argument that shows the introduction of F-heaps is useful; we establish the same bound for Karzanov's algorithm without needing any modification of the algorithm; we establish the $O(|S|^3+|S||T|)$ bound, for the case of bounded degree, using another modification of the MPM algorithm, but without the use of F-heaps.

Applications of these results

The importance of these algorithms and bounds are due to the common occurrence, both in practice and as problem models, of bipartite flow problems where $|S| << |T|$. The payoffs from our results are two-fold.

First, they lead to the fastest known solutions for certain combinatorial problems which can be modeled as bipartite flow. We illustrate this with two problems: for a well known scheduling problem, we reduce the previous best running time from $O(m^{2.5}n^3)$ to $O(mn^3)$, where m is the number of machines, and n is the number of jobs; and for a large class of 0-1 integer programming problems, we reduce the best upper bounds from $O((|S|^3|T|+|T|^2|S|^2)\log^2|T|)$ to $O(|S|^3|T|)$. In the second application, $|T|$ is often tremendously larger than $|S|$; the only bound on $|T|$ is $2^{|S|}$, although $|T|$ is polynomially bounded in $|S|$ for most of the interesting problems in this class.

Second, our results allow the use of bipartite flow as an efficient model for many problems that can be modeled both as bipartite flow and as flow in a non-bipartite graph. For most of these problems, the bipartite model is conceptually simpler, but the competing non-bipartite model typically has fewer nodes: $|S|$ versus $|S|^2$. For that reason, the belief reflected in the literature has been that the non-bipartite model permits faster execution of network flow algorithms-- $O(|S|^3)$ time verses the unattractive $O(|S|^6)$. The results of this paper show that flow in the competing bipartite models also can be solved in $O(|S|^3)$ time, and hence the accepted logic for preferring the more complex non-bipartite models is flawed, although we of course don't know which model ultimately permits the most efficient flow computation. In the applications section of this paper we will discuss several problems that have such competing bipartite and non-bipartite solutions. One such problem, the subgraph density problem, is also an example of the class of 0-1 integer programming problems mentioned above. Building on these results, we note that the time needed to solve weighted generalizations of the subgraph density problem can be reduced from the reported bound of $O(|n|^6)$ to $O(|n|^4)$.

More generally, our results encourage the consideration of bipartite flow models where more complex non-flow models are known, and demonstrate the utility of looking beyond general upper bounds.

Outline of the paper

In section 2 we give the basic definitions, review the MPM and Karzanov algorithms, and analyze the MPM algorithm applied to the bipartite flow problem. Using F-heaps and a new charging argument, we obtain the $O(|S|^2|T|)$ time bound. We also show the same time bound for Karzanov's algorithm. In section 3 we examine the case of bounded degree on each T-node (excluding the source or sink); we modify the MPM algorithm and show that it has running time $O(|S|^3+|S||T|)$ for this class of graphs. In section 4 we discuss applications of these results to a variety of problems: scheduling on uniform parallel machines, a class of 0-1 integer programming problems, the maximum subgraph density problem and weighted generalizations, network reliability testing, network vulnerability, statistical clustering, the transportation problem, decomposing a graph into disjoint spanning trees, optimal partitioning of a data-base, optimal rounding of statistical tables, security of statistical tables, and the sportswriter's end-of-season problem.

2. Computing maximum bipartite flow in time $O(|S|^2|T|)$

In this section we briefly describe and analyze the MPM maximum flow algorithm, with the modification that we use Fibonocci heaps [FT] in its implementation; this is the only modification we make to obtain the $O(|S|^2|T|)$ bound.

We will also show that Karzanov's algorithm without modification runs in $O(|S|^2|T|)$ time on a bipartite flow graphs. Our main contribution is the *analysis* of these algorithms on bipartite graphs.

2.1. The Modified MPM Algorithm

We assume familiarity with the s-t flow problem and the standard MPM algorithm but outline the algorithm in order to establish notation and introduce an important modification. The algorithm iterates *phases*, where a phase is the following:

Phase: Given a graph $G = (V,E)$ with source node s, sink t, and the current flow F, construct a directed *layered* graph G(F), and find the *maximal* s-t flow in G(F). A maximal s-t flow is one where at least one edge of every s-t path is saturated (a maximal flow is not necessarily a maximum flow). If the maximal flow is zero, then F is the *maximum* flow in G, else use the maximal flow to update F.

We will not describe how to construct a layered graph and update the flow, but it is well known [E] that both those operations can be done in time linear in the number of edges in the graph. In G(F) the nodes are partitioned into layers which are ordered so that edges only run from a layer to its successor. It is well known that the number of layers in the layered graph increases in each consecutive phase.

Finding a maximal flow: Now we describe in more detail how to find the maximal flow in a layered graph; our analysis of MPM will concentrate on these details. We need the following definitions.

Definition 2.1: For a node v in a layered graph G(F) define the *forward potential* of a node v, f(v) to be the sum of capacities on the edges out of v, the the *backward potential*, b(v) to be the sum of the capacities of the edges into v, and the potential p(v) = min[f(v),b(v)]. Define p = min[p(v)].

The algorithm finds a maximal flow in G(F) by iterating the following *augmentation* step:

augmentation: Find the node w of minimum potential, and push p(w) units of flow in G(F) from w to node t as follows. Choosing some ordering of the edges out of w, push flow along the edges in order, until p(w) units of flow are allocated. Note that at most one edge out of w is partially filled, i.e. has flow strictly between zero and its capacity. Edges with flow at their capacity are said to be *saturated*. For each node that receives incoming flow, push that flow out in the same way as above, and continue such pushes until p(w) units of flow reach node t. The facts that w is the node of minimum potential and G(F) is a layered graph assure that p(w) units will arrive at t. In a similar way, push p(w) units from w to s. After augmenting, all saturated edges are deleted. Notice that this either cuts all paths from s to w, or all paths from w to t, hence node w can be removed also.

When we change the flow in an arc we will call this a ***saturating edge push*** if the arc becomes saturated, otherwise we call it a ***non-saturating edge push***. Note that an augmentation changes the potential of all nodes incident with an arc whose flow is changed. In order to repeatedly find the minimum potential node efficiently, we will store and update the node potentials using the Fibonacci Heaps of Fredman and Tarjan [FT]. We will refer to these as F-heaps. The introduction of F-heaps here, instead of some other priority queue, is the only modification we make to the standard MPM algorithm.

Thus the work for each phase is:

(1) Build the layered network.
(2) Compute all node potentials and store them in an F-heap.

Iterate the following steps until a maximal flow is achieved.

(3) Find the node of least potential.
(4) Push flow through the network, updating the flow on changed arcs.
(5) Update the node potentials, and rebuild the heap of node potentials.

2.2. Analysis of The Modified MPM Algorithm

In the following analysis we will assume that $|T| > |S|$, however, a similar result will hold when $|T| < |S|$. Let E be the set of arcs in the flow network and let $V = S \cup T \cup \{s\} \cup \{t\}$, be the set of nodes in the network. Since G(F) is bipartite, every layer (except the $\{s\}$ and $\{t\}$ layers) consists exclusively of nodes of S or exclusively of nodes of T; we call a layer an S-layer or a T-layer depending on the type of nodes it contains.

Theorem 1: The algorithm described in section 2.1 finds a maximum flow in $O(|S|^2|T|)$ time on a bipartite flow network, for $|T| < 2^{|S|}$.

Note that the standard analysis would give $O((|S|+|T|)^3)$, so our analysis shows there is a substantial savings when $|T| >> |S|$. The assumption that $|T| < 2^{|S|}$ is no restriction in our applications, and is not needed in the analysis of Karzanov's algorithm.

Proof: We first prove a general lemma about algorithms which work in phases on a bipartite flow graph.

Phase Lemma: At most O(|S|) phases are required to find a maximum flow in a bipartite flow graph.

This is in contrast to the O(|S|+|T|) phases needed for a general graph.

Proof of phase lemma: In any layered graph G(F), every other layer (except {s} and {t}) is an S layer, hence G(F) has at most 2|S|+2 layers.□

We will now show that each phase can be run in O(|S||T|) time. Steps (1) and (2) are done once per phase and can be done in O(|E|) time, which is O(|S||T|). Using F-heaps, step (3) needs only time O(|V|log|V|) = O(|T||S|) for $|T| < 2^{|S|}$. The heart of the O(|S||T|) bound is in the analysis of steps (4) and (5). In order to analyze the work in step (4) we need to bound the number of edge pushes used. Each edge push either saturates the edge or is a non-saturating push. Clearly, the total number of saturating edge pushes in a phase is O(E) = O(|S||T|), so the key issue is how many non-saturating edge pushes there are in a phase. Standard analysis of MPM gives $O((|S|+|T|)^2)$. Our key observation is that this can be reduced to O(|S||T|). The proof of Theorem 1 will be completed with the following two lemmas.

Charging Lemma: Let A be the number of augmentations in a phase. Then the total number of non-saturating edge pushes in a phase is at most 4A|S| + |E|, which is O(|S||T|).

Proof: We examine the pushes from x, the minimum potential node, towards t. The pushes towards s are handled in a similar manner. In any augmentation there is at most one non-saturating edge push from any node. Each such non-saturating edge push will be **charged** to either an S-node or to a saturated arc.

Let (v,w) be an arc which has its flow increased without being saturated. If v is an S-node then (v,w) is charged to the node v. If v is a T-node then there must be some arc (u,v) that has its flow increased by the current augmentation, and u must be an S-node. If (u,v) is saturated by the current augmentation, then (v,w) is charged to (u,v). If (u,v) is not saturated, then (v,w) is charged to node u. Note that an arc (v,w) incident from the T-node v may be eligible to be charged to several arcs or nodes or both. In this case it can be charged to any one of them. Figure 1 illustrates an allocation of charges.

In each augmentation, each *S-node* can be at most charged at most twice: once for the arc leaving it which has a non-saturating edge push, and once for the arc leaving the T-node incident to this arc. Hence in A augmentations, there are at most 2A|S| charges to S-nodes per phase. In a phase, each *arc* can be charged at most once, and this can happen only in the augmentation which saturates it. So, there are O(|E|)=O(|S||T|) charges to arcs per phase. Thus the total number of charges per phase is at most 2A|S| + |E| = O(|S||T|), since A = O(|T|) and |E| = O(|S||T|). Repeating the analysis for pushes from x to s, and observing that every edge in the graph is saturated at most once, the bound of 4A|S| + |E| is proved. □

Potential Lemma: In a phase, the total number of node potential changes is at most 4A|S| + |E| = O(|S||T|), which bounds the time needed for step (5).

Proof: A node potential changes only when flow is pushed in an incident edge. Since potentials only decrease, the F-heap maintains the node potentials in time linear in the number of changes, which is, by the charging lemma, at most 4A|S| + |E| = O(|S||T|) per phase. Note that without the charging lemma, we would have to count $O((|S|+|T|)^2)$ for the potential changes, even using F-heaps. □

2.3. An $O(|S|^2|T|)$ bound for Karzanov's algorithm

In this section we will show that for bipartite graphs Karzanov's algorithm [HU], unmodified, has the same $O(|S|^2|T|)$, assuming only that $|S| \leq |T|$. We assume familiarity with Karzanov's algorithm, but outline it here in order to permit the analysis.

Karzanov's algorithm, like MPM, iterates phases, where each phase finds a maximal flow in a layered network by repeated augmentations. During a phase, the flow in an arc may be increased several times, but it can be reduced at most once, after which the arc is declared *closed* and can be used no longer. We need the following

Definition 2.2: A *preflow* is a function f on the arcs of a layered graph such that for each arc u $f(u) \leq \text{capacity}(u)$, and for any $v \in V$, $\sum_{e \in \alpha(v)} f(e) \geq \sum_{a \in \beta(v)} f(a)$, where $\alpha(v)$ is the set of arcs going into v and $\beta(v)$ is the set of arcs leaving v.

A node is *unbalanced* if the flow entering it is greater than the flow exiting it. A preflow is similar to an an ordinary flow, but it allows unbalanced nodes. Karzanov's algorithm repeatedly augments until there are no unbalanced nodes except s and t. An augmentation consists of two steps: *advance of preflow* and *balance of preflow*, which are described below. Let L_k denote the k-th layer of the layered graph, $k = 0, \ldots, r$, where $L_0 = \{s\}$ and $L_r = \{t\}$, and let j be the index of the layer from which the augmentation starts. Initially, all arcs are open, $j = 1$, and the arcs directed out of s are saturated.

Advance of Preflow: The purpose of this step is to establish a new preflow in the layered graph. Repeat the following operation for each L_i, $i = j, \ldots, r$, until either the last layer is reached or it is impossible to push any flow into the next layer:

For each unbalanced node $u \in L_i$, balance it, if possible, by pushing excess flow into the next layer.

Outgoing arcs are used in a fixed order, in such a way that the advance leaves at most one partially filled arc per node. Notice that it is not necessary to consider all the unbalanced nodes in a given layer; it suffices to look at those that have just had their incoming flow increased or their outgoing flow decreased.

Balance of preflow: Let L_k be the highest layer containing unbalanced nodes in the graph. For each unbalanced node $v \in L_k$, reduce the flow in its incoming arcs, starting from the last arc that was used, and proceeding in reverse order of usage until v is balanced. The flow in an arc is decreased only if the flow in other, more recently used, arcs has already been reduced to zero. Declare closed all arcs in $\alpha(v)$. The next advance starts at L_j, where $j = k - 1$.

We say a node v is *blocked* if every path from v to t contains at least one saturated arc. It can be proved [HU] that an unbalanced node becomes blocked after an advance, that a blocked node remains blocked throughout the entire phase, and that a node is balanced at most once in a phase. Furthermore, from the description of the algorithm, we see that every augmentation, except the last, balances at least one node. These properties, together show that Karzanov's method correctly computes a maximal flow in a layered graph using at most $|V| - 1$ augmentations per phase.

We summarize the steps in a phase of Karzanov's algorithm:

(1) Build the layered network.

Iterate the following steps until every node is balanced except s and t.

(2) Advance preflow.

(3) Balance the preflow.

Our analysis parallels that of MPM, the key argument being a modification of the charging lemma of section 2.1.

Theorem 2: Karzanov's algorithm finds a maximum flow in $O(|S|^2|T|)$ time, where $|S| < |T|$.

Proof: We will show that each phase can be carried out in $O(|S||T|)$ time, which, together with the phase lemma proves the theorem. Step (1) is done once per phase and takes $O(|S||T|)$ time. Steps (2) and (3) are repeated $O(|V|) = O(|T|)$ times. The total number of saturating edge pushes is $O(|E|) = O(|S||T|)$. The total number of edge flow reductions in both saturated and unsaturated arcs is also $O(|E|)$. The problem now is to bound the number of edges that are used in step (2) but not saturated, and which are not subsequently closed in step (3). For this analysis, we divide a phase into its augmentations. Each augmentation begins at a particular layer L and pushes flow from L to t. We charge all non-saturating edge pushes out of layers higher than L exactly the same way as in the Charging Lemma. However, the non-saturating edge pushes out of L must be charged differently.

In the first augmentation, $L = L_1$; so, we charge all non-saturating edge pushes out of nodes in L to saturated arcs leaving s. For each subsequent augmentation, excess flow is pushed out of a node $v \in L$ only if there was at least one arc out of v that had its flow reduced in the previous balance step. Thus, we charge a non-saturating edge push from v to the last arc emanating from v whose flow was reduced. Recall that such flow reduction is done at most once per arc in a phase, so there are at most $O(E)$ of these charges.

As was the case for MPM, all charges are either to S-nodes or to edges. During an augmentation, no S-node is charged more than twice, giving us $O(|S||T|)$ charges to S-nodes per phase. An arc is charged at most twice per phase, once when it is saturated and once when its flow is reduced; so, we have $O(|S||T|)$ charges to arcs. The total amount of work in each phase is thus $O(|S||T|)$, yielding an overall running time of $O(|S|^2|T|)$. □

2.3.1. Corollary to Theorems 1 and 2

Definition 2.3: A graph is called a *partitioned* flow network if its nodes can be partitioned into two sets S and I, so that that no two nodes in I are connected by an arc. The partitioned graphs properly contain the bipartite graphs, and for a bipartite graph either of the two sides of the graph can take the role of I.

Theorem 3: Both modified MPM and Karzonov's algorithm find a maximum flow in $O((|I|+|S|)|S|^2)$ time on a partitioned flow network.

Proof: At least one node from S must appear in any two consecutive layers except the first and the last two; so, there are $O(|S|)$ phases. Each non-saturating edge push can be charged to either a node of S or to a saturated arc, just as in the charging lemma. The argument holds for partitioned graphs because arcs incident to an I-node must be to or from nodes in S. Thus as in the bipartite case, a non-saturating edge push out of a node in I can be charged to an S-node or to a saturated arc. Thus the work within a phase is $O(|E|) + O(|V||S|)=O((|I|+|S|)|S|)$, and the total work is $O((|I|+|S|)|S|^2)$.□.

3. Bipartite flow with bounded degree on the T-nodes

In this section we consider the bipartite flow problem when the degree of each T-node of G, other than s or t, is bounded by the constant d. We will refer to this problem as the *bounded degree bipartite flow problem*. We show that a simple modification of the MPM algorithm, even without the use of F-heaps, results in a running time of $O(d|S|^3 + d^2|S||T|) = O(|S|^3 + |S||T|)$. Bounded degree bipartite flow problems are common; in fact, it is common that $d = 2$ (we will discuss several examples in the applications section). Note that since we have assumed that $|S| < |T|$, the case of bounded degree of S-nodes is trivial.

The Key Idea

The key idea in this section is that we will modify the definitions of node potentials and their use in the MPM algorithm so that, in each augmentation, we need search only the S-nodes for the node to which flow is pushed from s, and from which flow is pushed to t. Consequently, the number of augmentations inside a phase is bounded by |S| instead of |S|+|T|, and further, since we only need to find the minimum potential among the S-nodes, we can find the node in O(|S|) time by simply scanning all the S nodes. Hence no F-heaps are needed. The assumption of bounded degree on the T-nodes is used to allow efficient updating of the node potentials.

Definition 3.1: For a node y in a T layer of layered graph G(F), we define f(y) as before, as the sum of the capacities of all the edges out of y, and we define b(y) as the sum of the capacities of all the edges into y.

Definition 3.2: If (x,y) is a directed edge from an S-node x to a T-node y in G(F), and the capacity of (x,y) is c(x,y), then $\bar{c}(x,y)$ is defined to be Min[c(x,y), f(y)]. We call $\bar{c}(x,y)$ the *pseudo-capacity* of edge (x,y), and it is the amount that can be sent from x through edge (x,y) to the next S-layer closer to t.

Definition 3.3: For an S-node x in G(F), define f(x) (the forward potential) as $\underset{y \text{ in the layer after } x}{\Sigma \bar{c}}(x,y)$ i.e. the maximum amount of flow that can be pushed from x to all the nodes of the next S layer in G(F). Similarly, define b(x) (the backward potential) as the amount of flow that can be pushed from the S layer preceding the layer containing node v, to node v. Let p(v) = min[f(v), b(v)], and let p be the minimum p(v) value over all S-nodes of G(F).

Note that the definitions of potentials for a T-node are standard, but the definitions of potentials for an S-node differ from the standard definitions in that we are looking two layers ahead and back. Further, the definition of p differs from the standard in that it is the minimum pseudo-capacities during a phase. A push from an S-node x along edge (x,y) only affects the potentials of x and y and the capacity of edge (x,y). A push from a T-node y to a node w only affects the potentials of y, the capacity of (y,w), the potentials of the S-nodes adjacent to y, and the pseudo-capacities of the associated edges. Hence, since each T-node has bounded degree d, at most O(d) updates are required after any edge push. Then since the total number of edge pushes per phase is bounded by $O(|S|^2 + d|T|)$, the number of potential, capacity and pseudo-capacity changes is bounded by $O(d|S|^2 + d^2|T|)$.

Note that in order to find p at each augmentation, we need the current values of b(v) for each node v. These can be maintained in the same time bound that the forward potentials are, in a symmetric way. Hence the lemma is proved. □

Lemmas 3 and 4 and the phase lemma then yield the following

Theorem 3: With the above modifications of the MPM algorithm. bounded degree bipartite flow can be computed in time $O(d|S|^3 + d^2|S||T|) = O(|S|^3+|S||T|)$.

The case of d = 2: For d=2 (a common case) the modification of MPM can be viewed as merging length two paths between consecutive S-layers into a single edge. Then there are only |S| nodes in the resulting graph, and clearly then the phase runs in $O(|S|^2+|T|)$ time. No such simple interpretation is known for arbitrary values of d. We note that for $|E| = O(|T|)$, it is easily shown that the Dinic algorithm runs in $O(|S|^2|T|)$ time on bipartite graphs.

4. Applications

We will now present several problems which can be solved using network flow in a bipartite graph where $|T| >> |S|$.

4.1. Multiprocessor Scheduling with Release Times and Deadlines

We are given n jobs and m processors. The jth job has a release time r_j and deadline d_j such that the job cannot be started before time r_j and must be completed by time d_j. In addition the jth job has a processing time p_j which is the time required to complete the jth job. Each processor can run only one job at a time and each job can only be run on one processor at a time. We also assume that a job can be *preempted* at any time, and be resumed immediately on a new processor or later on any processor at no cost. Our goal is to find a schedule which completes all jobs subject to their release time and deadline restrictions.

Horn [HO] showed that this problem can be formulated and solved as a network flow problem as follows. Let $t_1 \leq t_2 \leq \ldots \leq t_k$ be the distinct times in the multiset $\{r_1, d_1, r_2, d_2, \ldots, r_n, d_n\}$. The time period between two successive t_i values will be called an *interval*. The flow network is used to determine how much of each job is to be completed within an interval. If an interval has length Δ, and q_j, $j=1,2,\ldots n$, is the amount of job j to be completed within this interval, then these processing amounts can be completed if and only if

(1) $\max\{q_j, j=1,2,..n\} \leq \Delta$, and

(2) $\sum_{j=1}^{k} q_j \leq \Delta m$.

Once the q_j values are determined, the schedule for this interval can be constructed in O(n) time [MC].

To find the processing amounts within an interval we construct a flow network with a node for each job and a node for each interval. The source is connected to the jth job node with an arc of capacity p_j. The jth job node has an arc to the ith interval node if $r_j \leq t_i$ and $d_j \geq t_{i+1}$, and this arc has capacity $t_{i+1} - t_i$. The ith interval node has an arc to the sink with capacity $m(t_{i+1} - t_i)$. See figure 2. The flow network described is bipartite, with $|S|=n$ and $|T|=k$. By Theorem

1 and the symmetric roles of S and T, we can find a maximum flow in $O(|S||T|^2)=O(nk^2)$ time. This is an improvement on the previous $O(n^3)$ time bound in the common case when k is $o(n)$.

4.2. Uniform Processors

We will now generalize the previous scheduling problem by allowing the speeds of the processors to be different. Let the m processors have speeds $s_1 \geq s_2 \ldots \geq s_n$. A job with processing requirement p takes p/s_i units of time to be completed on the ith processor. As in the previous problem, once we have determined the amount of processing to be done on the jobs within an interval, the schedule for that interval can be constructed quickly. Gonzalez and Sahni [GOSA] have shown that if $q_1 \geq q_2 \ldots \geq q_n$ are the amounts of processing to be done in an interval of length Δ, then a schedule can be constructed in O(nlogm) time if and only if

3) $\sum_{j=1}^{i} q_j \leq \sum_{j=1}^{i} s_j \Delta$, for each $i = 1,2,\ldots,m-1$ and

4) $\sum_{j=1}^{n} q_j \leq \sum_{j=1}^{m} s_j \Delta$

Federgruen and Groenevelt [FeGr] have shown that the amount of processing to be done within each interval can be found using the following flow network. As in the case for processors with identical speeds, there will be a source and n job nodes. However, we will have m *speed* nodes, (i,1), (i,2), ..., (i,m), associated with the ith interval. The source is connected to the jth job node with an arc of capacity p_j. The jth job node is connected to node (i,k) if $r_j \leq t_i$ and $d_j \geq t_{i+1}$. This arc has capacity $(s_k - s_{k+1})(t_{i+1} - t_i)$, (where we let $s_{m+1}=0$). Each speed node (i,r) is connected to the sink with an arc of capacity $(s_r - s_{r+1})(t_{i+1} - t_i)r$.

The number of intervals k can be as large as 2n-1; so, there are O(mn) speed nodes. A simple analysis of solving this network using the MPM algorithm results in an $O(m^3n^3)$ time bound. Federgruen and Groenevelt reduced this to $O(m^{2.5}n^3)$ using Cherkaski's flow algorithm [Cher].

However, since we have a bipartite flow network with $|S|=O(n)$ and $|T|=O(mn)$, Theorem 1 gives us a time bound of $O(|T|\ |S|^2)=O(mn^3)$. In fact, Federgruen and Groenevelt show that if there are only r distinct processor speeds we can reduce the number of speed nodes to $O(rn)$ which gives us a time bound of $O(rn^3)$. Hence the bounds obtained using Theorem 1 of this paper beat the best previously established bounds.

4.3. Applications based on Provisioning

A rich class of problems that can be modeled and solved as bipartite flow fall under the heading of *provisioning* or *shared fixed cost* problems. Picard [PIC79-1,PIC82-2] has shown that a large class of 0-1 integer programming problems can be solved this way. These problems are solved with a sequence of up to $|S|$ bipartite flow computations, each on a bipartite graph with $|S| \leq |T| \leq 2^{|S|}$. In that graph, each S node represents a variable in the integer programming problem, and each T-node represents a constraint. In general, $|T| >> |S|$, although in most of the applications, the number of T-nodes is polynomial in $|S|$. For the class of problems defined in [PIC79-1,PIC82-2], Picard obtains the general time bound $O((|S|^3|T|+|T|^2|S|^2)\log^2|T|)$ using complex flow algorithms. Theorem 1 in this paper shows that these problems can all be solved by MPM with F-heaps, or by Karzanov's algorithm in time $(|S|^3|T|)$. Below we will discuss one of the specific problems discussed in [PIC79-1,PIC82-2]: the maximum subgraph density problem.

4.3.1. Maximum Subgraph Density

Let $G = (V,A)$ be an undirected graph without parallel edges, where $|V| = n$, and $|A| = m$. Let $V' \subset V$ be a subset of vertices of V, and let $G(V')$ be the subgraph of G induced by V'. The *density* of $G(V')$ is defined as the number of edges of $G(V')$ divided by the number of nodes in V'. The maximum subgraph density problem is to find a set $V' \subset V$ to maximize the density of $G(V')$. The problem is of interest for its own sake, and has non-direct applications in problems of network vulnerability [G83], [CUNN83], statistical clustering [M72, M77], and decomposition of graphs into edge disjoint spanning trees [PIC79-2], [TU] [NW]. Several generalizations of subgraph density involving weights on edges and nodes are introduced in [GOL]; we will discuss some of these below.

Surprisingly, the subgraph density problem can be solved efficiently, despite its similarity to the maximum clique problem. All of the proposed methods involve solving at most $|V|$ optimization problems called parametric problems; each has the following form: Given graph G = (V,A) and a number λ, find the subset of vertices $W \subset V$ to minimize $\lambda|W| + |c(A(W))|$, where c(A(W)) is the set of edges in A-A(W), and A(W) is the set of edges in the subgraph G(W) induced by W. The solution to the subgraph density problem is found by searching for the value of λ where the solution to the parametric problem is exactly $|A|$. Details of how the subgraph density problem reduces to these parametric problems are given in [PIC79-1,PIC82-2,GOL] and omitted here. Each of the above parametric problems can be solved using network flow on either a bipartite or a non-bipartite graph. In either case, exactly the same sequence of parametric problems is solved. We will first describe a bipartite solution to the parametric problem found in [PIC79-1,PIC82-2], and then a non-bipartite solution derived from [CUN83]. Other non-bipartite solutions appear or can be derived from [PIC82-1], [GOL], [FUG], [TOP].

In the bipartite solution, the graph G(S,T) is constructed from G as follows. Node set S contains one node for every node in G, and node set T contains one node for every edge of G. A node u in S is connected to a node e in T if and only if u is an endpoint of the edge in G that e represents; each edge from S to T has infinite capacity. Node s is attached to each node in S, and each of these edges has capacity λ. Node t is attached to each node in T and each of these edges has capacity one. Then, the minimum s-t cut in G(S,T) defines the optimal node set W: W contains every S node that is on the t side of the minimum s-t cut (see figure 3).

In the non-bipartite solution of the parametric problem, the graph NB is constructed by adding a source node, s, and a sink node, t, to the original graph G. There is one edge (s,v) connecting s to each node v in V; this edge has capacity λ. There is also one edge (v,t) connecting each node v to t; each such edge (v,t) has capacity d(v)/2, where d(v) is the degree of node v in G. Each original edge in G has capacity 1/2 (see figure 4). The optimal node set W is again the set of nodes of S that are on the t side of a minimum s-t cut.

It has been generally accepted in the literature that the subgraph density problem can be more efficiently solved using the non-bipartite graph NB than by using the bipartite graph G(S,T). Since, in the worst case, |A| is $\Theta(|V|^2)$, naive application of the general upper bounds on the running times of the Dinic, MPM and Karzanov algorithms on G(S,T) yield guaranteed times (for each flow computation) of no better than $O(|V|^8)$, $O(|V|^6)$, and $O(|V|^6)$ respectively. However, when run on NB, the corresponding upper bounds are the square root of the above bounds for G(S,T). Even using more complex algorithms [SL] which take advantage of the sparsity of G(S,T) (G(S,T) is sparse even when $|A| = \Theta(|V|^2)$), naive application of the upper bounds gives a guaranteed worst case running time of $O(|V|^4 \log|V|)$ for the bipartite graph; hence even sophisticated algorithms appear to run slower on the bipartite graph than do the simpler algorithms on the non-bipartite graph.

The results of section 3 refute the above reasoning. In G(S,T) every node in T has bounded degree two, and so both the Dinic and the modified MPM algorithms run as fast ($O(|V|^4)$, $O(|V|^3)$) on G(S,T) as those algorithms run on NB. Further, it can be shown, using ideas similar to those in this paper, that the unmodified Karzanov algorithm also runs as fast ($O(|V|^3)$) on G(S,T) as it does on NB. Hence, the accepted argument for the superiority of the non-bipartite model is incorrect, although we don't know which model actually permits more efficient solutions.

4.3.2. Weighted Subgraph Density Problems

Several generalizations of the subgraph density problem were proposed in [GOL]; we mention here an observation that reduces the running time of the solutions. The most general density problem is that each edge e has a (real) weight w(e) and each node v has a (real) weight w(v), and the objective is to find a subset V′ of the vertices to maximize the ratio w(A(V′))/w(V′), where w(V′) is the total weight of the vertices in V′, and w(A(V′)) is the total weight of the edges in the graph induced by V′. Note that the subgraph density problem is just the problem where the weight of each edge and vertex is one.

The weighted density problem is again solved with a sequence of parametric optimization problems, each of the form: given λ, find the subset of vertices $W \subset V$ to minimize $\lambda w(W) + w[c(A(W))]$. Each of these problems is solved by network flow in a graph identical to G(S,T) or NB, except for the edge capacities; in G(S,T), each edge (s,v) has capacity $\lambda w(v)$, and each edge (e,t) has capacity w(e).

The generalized density problem is solved by finding the value λ^* of λ for which the parametric problem has solution w(A). The key issue then is how to vary λ to home in on λ^*. Goldberg [GOL] gives a method which never uses more than $O(|V|^3)$ trial values of λ, and hence finds the optimal V′ in $O(|V|^6)$ arithmetic operations (on reals). We note here that at most $|V|$ trial values of λ are ever needed, and hence the weighted density problem can be solved in $O(|V|^4)$ time. This follows from a result in [ES] that as λ varies from 0 to infinity, there can only be $|V|$ values of λ where the set of edges in the minimum cut changes. Further, the value of the minimum cut as a function of λ can be completely determined (traced out) in $O(|V|^4)$ time [ES]; after that, λ^* can be found trivially. A similar theorem [ST] applies to minimum cuts in the non-bipartite graph NB. Other weighted problems (over rationals) are also discussed in [GOL] and their solutions can similarly be sped up with the above observation.

4.4. Additional Applications

We briefly sketch several more applications of flow in a bipartite graph with node sets S and T, where $|T| >> |S|$ either as a structural feature of the model (as in subgraph density) or in frequent problem instances

1. Network reliability Testing [SAS]: In a communication network, each node i can test k(i) incident lines per day, and each line j must be tested t(j) times. Minimize the number of days to finish the tests. Here $|T| = \Theta(|S|^2)$, and $d = 2$.

2. Find a maximum size set of edge disjoint spanning trees in an undirected graph [TU], [NW], [PIC79-2]. $|T| = \Theta(|S|)^2)$, and $d = 2$.

3. Network vulnerability [G83], [CUNN83]: Given a connected graph G, and a set of edges W in G, define c(W) as the number of additional components created by deleting edge set W from G. Find the best edge set W to maximize the ratio $c(W)/|W|$. Here again, $|T| = \Theta(|S|^2)$, and d = 2.

4. Partitioning a data-base between fast and slow memory [ES]: The data base consists of n pieces of data which will be distributed between fast (expensive) memory and slow (cheaper) memory. There are m possible queries that will be made on the data base, each defined by a subset of the data. A cost c(i) is incurred for storing data i in fast memory, and a cost c(j) is incurred when query j is made and some of the data needed for query j is in slow memory. Determine the minimum cost partition of the data. $|T| = m$ and $|S| = n$, and typically $m >> n$.

5. Security of statistical tables [G84]: Estimate the best upper and lower bounds on the value of any missing data in an n by m statistical table. $|T| = m$ and $|S| = n$, and m and n can be very different.

6. Provisioning and shared fixed cost [RHY]: general selection problem with many specific applications of bipartite flow.

7. Optimal controlled rounding of statistical tables [CE]: Given a two dimensional table D of cross tabulated integer statistics, and numbers d and p, round each entry to a multiple of d so that the sum of the rounded numbers equals their rounded sum, and so that the $\Sigma|D(i) - R(i)|^p$ is minimized, where D(i) and R(i) are the original and rounded values in the table.

8. The classical transportation problem [FF]: Given n origins, m destinations, a supply s_i at each origin i, a demand of d_j at each destination j, and a capacity c(i,j) on the amount that can be shipped from origin i to destination j, determine if all demands can be satisfied by the supplies, and determine how to allocate the supplies. Here n and m can be very different. The classical problem of disjoint set representatives is a special case of this problem.

9. Sportswriter's End-of-Season Problem [SW],[HOFF]. Given the win-loss record of a set of n teams in a league, and the schedule of the remaining games, determine whether there is an outcome of the remaining games so that a given team x wins the pennant[3] , (wins more games than any other team). Here again, $|T| = \Theta(|S|^2)$ and $d = 2$.

5. Acknowledgement

We would like to thank Gregory Sullivan at Yale for helpful and insightful comments during the middle stages of this work.

[3]A harder problem of computing the probability that x wins is NP-hard [GS]

6. References

[CH] Cherkaski, B., "Algorithm of construction of maximal flow in networks with complexity $O(|V|^2|E|^{1/2})$ operations",Math. Methods of Solutions of Economic Problems 7 (1977), 117-125 (in Russian).

[CE] L. Cox and L. Ernst, "Controlled Rounding", INFOR vol. 20, no. 4, Nov. 1982.

[CUNN83] W. H. Cunningham, "Optimal Attack and Reinforcement of a Network", Preprint Dec. 1983.

[E] S. Even, Graph Algorithms, Computer Science Press, 1979.

[ES] M.J. Eisner and D.G. Severance, "Mathematical Techniques for Efficient Record Segmentation in Large Shared Databases", J. of the Assoc. for Comp. Mach. 23, No 4, 619-635, 1976.

[FF] L. Ford, and D. Fulkerson, Flows in Networks. Princeton University Press, 1962.

[FG] A. Federgruen and H. Groenevelt, "Preemptive scheduling of uniform machines by ordinary network flow techniques", preprint.

[FT] M.L. Fredman and R.E. Tarjan, "Fibonacci Heaps and their uses in improved network optimization". 25th Annual Symposium on Foundations of Computer Science. IEEE. 1984.

[FU] "Lexicographically optimal base of a polymatroid with respect to weight vector", Math. of Operations Research. 5 (1980).

[GOL] A.V. Goldberg, "Finding a Maximum Density Subgraph", Tech. Report UCB/CSD 84/171 May 1984, Computer Science Division, U.C. Berekely.

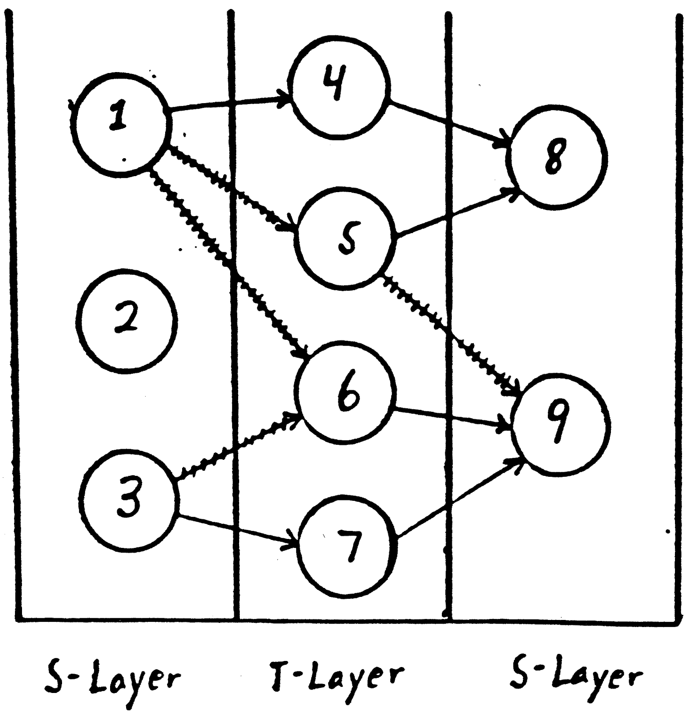
[GOSA] T. Gonzalez and S. Sahni, "Preemptive scheduling of uniform processor systems", JACM 25, no. 1, Jan. 1978, pp. 92-101.

[G84] Gusfield, "A Graph Theoretic Approach to Statistical Data Security", Yale Technical Report no. 326, August 1984.

[G83] Gusfield, "Connectivity and edge disjoint spanning trees". Information Processing Letters, February 26, 1983.

[GS] Gusfield and G. Sullivan, "Determining Possible End-of-Season Rankings". Yale Technical report in preparation.

[HO] Horn, W., "Some simple scheduling algorithms", Naval Res. Log.



Arc	Charged to
(1,4)	node 1
(3,7)	node 3
(4,8)	node 1
(5,8)	arc (1,5)
(6,9)	arc (3,6)
(7,9)	node 3

Only augmented arcs are shown. An arc $\rightsquigarrow$ is saturated by the current augmentation.

Figure 1 . Charge allocation for non-saturating edge pushes.

Job	p_j	r_j	d_j
1	4	0	6
2	8	0	14
3	3	2	5

$n = 3$
$m = 2$

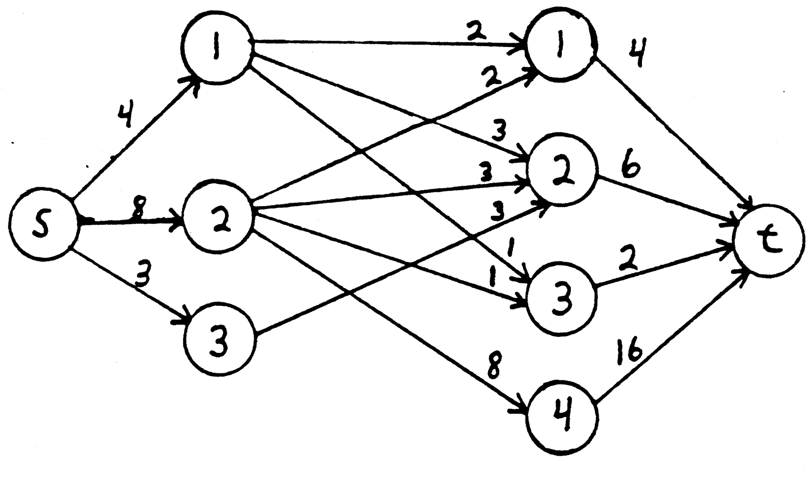


Figure 2 . A flow network for a 2-processor scheduling problem.