Degree-Constrained Network Flows

Patrick Donovan
School of Computer Science
McGill University
patrick.donovan@mail.mcgill.ca

Bruce Shepherd
Department of Mathematics and Statistics
McGill University
bruce.shepherd@math.mcgill.ca

Adrian Vetta
Department of Mathematics and Statistics, and
School of Computer Science
McGill University
vetta@math.mcgill.ca

Gordon Wilfong
Mathematical and Algorithmic Sciences Center
Bell Laboratories
gtw@research.bell-labs.com

ABSTRACT

A d-furcated flow is a network flow whose support graph has maximum outdegree d. Take a single-sink multicommodity flow problem on any network and with any set of routing demands. Then we show that the existence of feasible fractional flow with node congestion one implies the existence of a d-furcated flow with congestion at most $1+\frac{1}{d-1}$, for $d \geq 2$. This result is tight, and so the *congestion gap* for d-furcated flows is bounded and exactly equal to $1+\frac{1}{d-1}$. For the case $d=1$ (confluent flows), it is known that the congestion gap is unbounded, namely $\Theta(\log n)$. Thus, allowing single-sink multicommodity network flows to increase their maximum outdegree from one to two virtually eliminates this previously observed congestion gap.

As a corollary we obtain a factor $1+\frac{1}{d-1}$-approximation algorithm for the problem of finding a minimum congestion d-furcated flow; we also prove that this problem is maxSNP-hard. Using known techniques these results also extend to degree-constrained unsplittable routing, where each individual demand must be routed along a unique path.

Categories and Subject Descriptors

F.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Algorithms, Theory

Keywords

Approximation algorithm, multicommodity flows, network flows, confluent flows

STOC'07, June 11–13, 2007, San Diego, California, USA.

1. INTRODUCTION

We consider the single-sink multicommodity network flow problem. We have a directed network (graph) $G=(V,A)$ with sink node t. Each node $v \in V$ wants to route r_v units of flow to the sink; this is termed the *demand* of node v. Furthermore, each node $v \in V$ has a fixed uniform capacity (by scaling we may take this capacity to be 1 as we allow fractional r_v values). Our interest lies in examining *bounded degree* or *degree-constrained flows*, that is, feasible flows whose support graphs have bounded outdegree[1] at every node. Since we only examine node congestion problems we assume our graphs have no parallel (that is, in the same direction) arcs or loops. Consequently, the maximum outdegree of any node is less than n, where n is the number of nodes in the network. We classify such flows as follows. The class of network flows with outdegree at most d is denoted by $\mathcal{C}_d$; we call such flows *d-furcated*. The cases $d \in \{1,2,\infty\}$ are of particular interest to us[2]. These flow classes are:

- $\mathcal{C}_\infty$ (*Fractional Flows*): flow from a node v to the sink t may be routed fractionally along *any* path; in particular, v may send flow on any number of outgoing arcs.
- $\mathcal{C}_1$ (*Confluent Flows*): flow from v to t must be routed on a *unique* path; in particular, v sends flow on at most one outgoing arc.
- $\mathcal{C}_2$ (*Bifurcated Flows*): flow may be sent from v on at most two outgoing arcs.

Bounded degree flows are natural and elegant combinatorial objects in their own right. Interest in them, however, is primarily motivated by certain distributed routing protocols. For example, consider how confluent flows are produced by the *open shortest path first* (OSPF) protocol. This protocol is essentially a distributed implementation of Dijkstra's algorithm. Consequently, for a specific network destination t, it populates each router v's next hop entry for t with some neighbour u of v for which there is a shortest path from v to t through u. In this context "shortest" is determined with respect to some costs on the links (arcs) and, in intra-domain networks, these costs may be altered by the

[1]Note that we make no restriction on the indegree of a node.
[2]Note that by assumption, we have $\mathcal{C}_\infty = \mathcal{C}_{n-1}$

network operator to achieve better traffic flow through the network (see, for example, [8]). Hence, under these constraints the collective flow destined for t is routed along a directed arborescence (rooted at t); that is, we have a confluent flow.

In most intra-domain networks, however, flows with higher but bounded degrees are allowed. For example, if there is more than one "shortest path" from v to t, operators may place two or more next hops for t in the routing table. Traffic to t is then typically split using a round-robin approach (also cf. [7] where alternatives to the round-robin scheme are considered). This motivates the present work. We wish to develop some of the network flow theory underlying the basic question: what happens if we allow multiple next hops per destination in our routing tables? In particular, how does network performance improve as the permissible outdegree increases? To answer this question we must adopt some performance measure to compare our various traffic flows. A variety of such measures could be used; we follow the approach of [3], [2] where the performance measure applied is worst case congestion of a node. The *congestion* of a node is its load divided by its capacity, where the *load* of a node v is just the total flow value through v (this includes the demand of v itself). Hence, in a uniform capacity network we wish to minimise the maximum load of a node.[3]

1.1 Our Results and Previous Work

Our goal is to assess the cost, in terms of congestion, of restricting network flows to only route on a bounded number of arcs out of any node. Specifically, what is the cost of routing using the confluent or d-furcated flow constraint? As in [3], [2], we consider uniform node capacity (equivalently, *uncapacitated*) networks exclusively. Suppose that G contains a fractional flow satisfying all the demands in which no node has load more than 1. One may then ask, is there a d-furcated flow that routes all the demands and has low congestion at every node?

Therefore we are interested in the *congestion gap*, $\gamma(d)$, of a flow class $\mathcal{C}_d$, which is the worst ratio, over any network and any set of demands, of the congestion of an optimal flow in $\mathcal{C}_d$ to the congestion of an optimal flow in $\mathcal{C}_\infty$. This question was first considered by Chen, Rajaraman and Sundaram [3], who showed there always exists a confluent flow with congestion $O(\sqrt{n})$. This was subsequently improved by Chen et al. [2] who proved a congestion bound of $O(\log n)$ and gave an example to show that this result is tight. (More precisely, they gave a bound of $O(\log k)$ where k is the number of nodes with outgoing arcs to the sink.) Hence the congestion gap $\gamma(1)$ between fractional flows and confluent flows is $\Theta(\log n)$ in an uncapacitated (i.e., uniform capacity) network. Thus, the gap between flows in $\mathcal{C}_1$ and flows in $\mathcal{C}_\infty$ is unbounded but, evidently, as the maximum outdegree of a flow is allowed to increase, the congestion gap tends to one. However, it was not known whether obtaining a bounded congestion gap required allowing an unbounded maximum degree. Perhaps surprisingly, we prove that a bounded congestion gap can be obtained with bounded outdegrees. In fact, the congestion gap is all but eliminated if we allow for bifurcated rather than confluent flows: Given a fractional flow of congestion one, there is a bifurcated flow with congestion at most two. Thus, our main result is that the congestion gap $\gamma(2)$ between flows in $\mathcal{C}_2$ and flows in $\mathcal{C}_\infty$ is at most two in uncapacitated networks. We also show that this bound is tight. Moreover, our techniques show the rate at which the congestion gap is eliminated as d grows; the congestion gap $\gamma(d)$ between d-furcated flows and fractional flows is at most $1 + \frac{1}{d-1}$.

Our proof is algorithmic and so provides a factor 2 (respectively, factor $1 + \frac{1}{d-1}$) approximation algorithm for finding a minimum congestion bifurcated (respectively, d-furcated) flow in a single-sink multicommodity flow problem. Finally, we show that this problem is maxSNP-hard.

1.2 Overview of the Algorithm

Before giving a formal description of the algorithm, it may be useful to outline our basic approach. The algorithm starts with a fractional single-sink multicommodity flow whose maximum node load is 1. Our objective is to transform this into a d-furcated flow with maximum load at most $1 + \frac{1}{d-1}$. We proceed by applying a number of operations that gradually bring the flow into a more manageable form. Some of these operations are well-known. For instance, we may reduce the flow on any directed cycle until the support of our flow is acyclic. Second, if there is a node v for which flow only leaves v on a single arc (u, v), then we contract this arc. A third operation applied is one used in [4] and coined a *sawtooth cycle* augmentation in [2]. A (general) sawtooth cycle consists of any cycle (in the underlying undirected support graph) that has the form $\{(u_0, v_0), P_0, (u_1, v_1), P_1, (u_2, v_2), P_2, \ldots, (u_r, v_r), P_r\}$. Here each P_i consists of a directed path from u_{i+1} to v_i (subscript arithmetic modulo $r + 1$). Given such a cycle, observe that we may increase the flow on each (u_i, v_i) by some $\epsilon > 0$, and decrease the flow on each P_i by ϵ, without increasing the load at any node. In [4, 2], such cycles are considered at the "frontier" of the graph, that is, the nodes v_i are all neighbours of the sink node t. Here we consider sawtooth cycles that may bounce around throughout the graph. One contribution of the paper is a structural result that shows if there is no such sawtooth, then the directed support graph has a very nice structure: the graph consists of a collection of induced trees with the property that each vertex is in at most two trees (see Theorem 3.4).

We defer a precise description but remark that with this structure, the support resembles a "layered graph". A sketch of the final stage of the algorithm is then as follows. Given our layered graph, we define the layers as follows: $L_0 = \{t\}$ and for each $i \geq 0$, $L_{i+1} = \{u \notin \cup_{j \leq i} L_j : \exists\, (u, v) \in A, v \in L_i\}$. Here the graph induced by adjacent layers $L_i \cup L_{i+1}$ is a forest corresponding to the aforementioned induced trees. Moreover, each node in L_{i+1} has outdegree at least two. This allows us to process our flow greedily from "top to bottom". That is, we first adjust the flow out of L_{max}, then out of L_{max-1}, etc. At each stage we also consider the nodes in level L_{i+1} greedily by picking the most remote nodes in the current forest, that is, the nodes which are adjacent to at least one leaf in the forest. At a suitably chosen node, we show how to redirect its flow without causing the congestion at any node in the next layer, L_i, to exceed $1 + \frac{1}{d-1}$.

[3]Congestion can also be defined in terms of congestion along a link. Note that for confluent flows, the maximum load on an arc in the network must occur on some link into the destination t. Thus if all link and node capacities are 1, then the worst case node and link congestion problems are identical.

2. THE ALGORITHM: PHASE I - FLOW SIMPLIFICATION

We now describe our approach in detail. We may remove t and call its set of in-neighbours $\Gamma^-(\{t\}) = \{t_1, t_2, \ldots, t_k\}$ the *sink nodes*. The goal of each node is then to route its demand to any combination of the sinks. We begin with a fractional flow f in G satisfying all the demands and having maximum node load 1. Clearly, we may assume that the (support of the) flow is acyclic; that is, the set of arcs with non-zero flow induce an acyclic graph. Our goal is to convert f into a d-furcated flow f' with node congestion at most $1 + \frac{1}{d-1}$. To achieve this we apply a two-phase algorithm. In the first phase we perform a combination of operations that simplify the structure of flow. Given this simplified structure, in the second phase, we redirect parts of the routing to obtain a d-furcated flow.

Given the initial fractional flow f, we begin by applying the following two operations.

- **Contractions.** If v is a node with outdegree 1 we contract the arc (v, u), where u is the out-neighbour of v. The demands of u and v are assigned to the new contracted node. This gives a new flow whose support contains one less node than the original flow.
- **Breaking Sawtooth Cycles.** A sawtooth cycle is a collection

  $$\{(u_0, v_0), P_0, (u_1, v_1), P_1, (u_2, v_2), P_2, \ldots, (u_r, v_r), P_r\}$$

  where (u_i, v_i) is an arc and P_i is a directed path from u_{i+1} to v_i (subscripts modulo $r + 1$). Note that reversing the arcs in every P_i would produce a directed cycle. (Note also that $r = 0$ is allowed, and since we assume G has no parallel arcs, P_0 would have length at least 2 in this case.) Now we may "augment" along such a cycle by adding ϵ flow to each arc (u_i, v_i) and subtracting ϵ flow from every arc in each P_i. This gives a new flow that still satisfies the capacity constraints of every node on the cycle. Therefore, we *eliminate* or *break* this cycle by choosing ϵ to be the minimum flow on one of the arcs in any of the P_i; this gives a new flow whose support contains one less arc than the original flow.

Note that performing a contraction operation may produce new sawtooth cycles. It follows that an arc of the form (u_i, v_i) may, in fact, correspond to a path in the original network whose internal nodes have outdegree one (and, hence, have been contracted into their out-neighbour). Observe that, in this case, load on these nodes may actually increase when we eliminate the sawtooth cycle. However, the flow still obeys the capacity constraints because these internal nodes have *less* flow than the end node of the path v_i and that node is not overloaded after augmenting a sawtooth cycle.

Eliminating a sawtooth cycle may also reduce the outdegree of a node to one causing more contractions. Thus we continue to perform either operation until no such operations are possible. Clearly, we perform only a polynomial number of operations as we either reduce the number of nodes or the number of arcs by one in each step.

Thus at the end of the first phase of the algorithm we obtain a fractional flow f which contains no sawtooth cycles and no nodes with outdegree 1. In order to proceed further, the key is to understand the structure of graphs that do not contain sawtooth cycles. Accordingly, before presenting the second phase of the algorithm, we next obtain a characterisation of graphs without sawtooth cycles. We remark that this characterisation gives an efficient way to detect sawtooth cycles; consequently, Phase I of the algorithm can easily be implemented in polynomial time.

3. SAWTOOTH CYCLES AND ACYCLIC DIGON-TREE REPRESENTATIONS

We now examine the structure of the flow obtained at the end of Phase I.

3.1 An Auxiliary Digraph

To commence we first give a simple characterisation of when a directed graph G contains a sawtooth cycle using an *auxiliary digraph* $D(G)$ (or simply D if the context is clear). The auxiliary digraph D is bipartite with (node) bipartition classes $\ominus$ and $\oplus$; for each node $v \in G$, we have two nodes $v^- \in \ominus$ and $v^+ \in \oplus$ in D. The auxiliary graph contains three types of arc (see Figure 1).

1. For each node $v \in G$, there is an arc from v^- to v^+ in D; we call this a *node arc.*
2. For each arc (u, v) in G, we have an arc from u^- to v^+ in D; we call this a *real arc.*
3. For each arc (u, v) in G, we have an arc from v^+ to u^- in D; we call this a *complementary arc.*

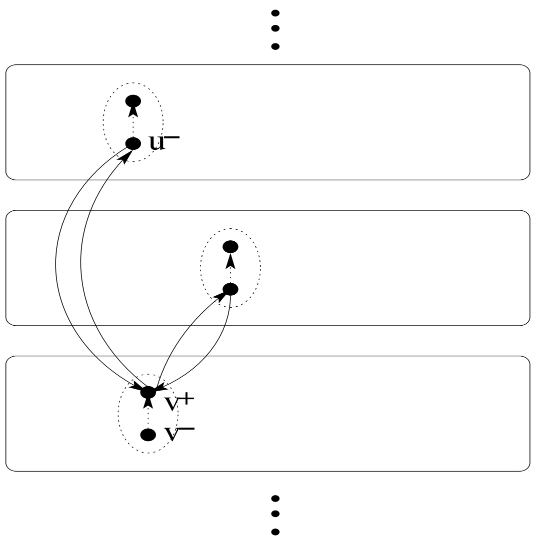

Figure 1: The auxiliary digraph.

Therefore D consists of a set of digons[4] (formed by real and complementary arcs) plus a collection of node arcs. Whether or not G contains a sawtooth cycle is then simply determined by the maximum length of a cycle in the auxiliary graph.

[4] A *digon* is a pair of arcs that form a simple directed cycle of length 2.

THEOREM 3.1. *G contains a sawtooth cycle if and only if the auxiliary graph D contains a (simple) cycle of length at least three (and hence at least four).*

PROOF. Take a cycle C in the auxiliary graph D that is not a digon (2-cycle). Since D is bipartite, this cycle has the form $C = \{v_1^-, v_2^+, v_3^-, v_4^+, \ldots, v_{2s-1}^-, v_{2s}^+, v_1^-\}$, where the v_i's represent nodes in the original network G. Note that we have $v_{2i-1} = v_{2i}$ in the cases where a node arc for v_i is used. Observe that by construction:
(i) a node arc in C must be followed by a complementary arc.
(ii) a real arc in C must be followed by a complementary arc.
(iii) a complementary arc in C may be followed either by a real arc or by a node arc.
It follows that C can be partitioned into blocks that consist of a real arc followed by an odd length path which has alternating complementary arcs and node arcs. Now observe that these odd length alternating paths correspond to the complements of real paths in G. Therefore C must contain at least one real arc a otherwise it consists just of the odd length alternating path contradicting the bipartiteness of D. Since C is not a digon in D, the odd length alternating path does not consist simply of the complement of a. Thus, C identifies a sawtooth cycle in G. Similarly, a sawtooth cycle in G corresponds to a cycle of length at least 4 in D. □

Next we show how to determine whether any digraph has a cycle of length at least 3. This leads to a characterisation for the existence of sawtooth cycles in terms of induced trees in the graph; this, in turn, is used to orchestrate the second phase of the algorithm.

3.2 Digon-Tree Representations

First we define the concept of a digon-tree representation. Suppose $D = (V, A)$ is a directed graph such that the underlying graph (edge-)induced by the digons of D forms a forest. Each component of this forest is called a *digon-tree*. Then we say that D has a *digon-tree representation* $\mathcal{D}$ where $\mathcal{D}$ is the digraph obtained by contracting (and then deleting) the digon edges of D. If $\mathcal{D}$ is acyclic, then we say that D has an *acyclic* digon-tree representation. The nodes of $\mathcal{D}$ are called *digon-tree nodes*, and a node which is not incident to a digon in D, forms a *singleton* digon-tree node in $\mathcal{D}$.

Digon-tree representations characterise when there are no circuits of length greater than 2, i.e., when every circuit in a digraph contains a digon. These representations are used in the second phase to guide how flow is redirected; in particular, we use them to determine the order in which nodes have their outgoing flow redirected. Note that if every circuit contains a digon, then the graph contains no loops or cycles of length at least three. The following appears in [9].

THEOREM 3.2. *Let D be a digraph without loops. Then D has no cycle of length at least three if and only if it has an acyclic digon-tree representation.*

PROOF. ($\Rightarrow$) Suppose D contains no loops or cycles of length at least three. If the underlying graph induced by the digons in D contains a cycle then clearly we have a cycle of length at least three, a contradiction. So the digons induce a forest F. Thus, D has a digon-tree representation $\mathcal{D}$.

Now assume that $\mathcal{D}$ contains a loop a at digon-tree node T. The arc a cannot correspond to a loop in D, or by definition of $\mathcal{D}$ to an arc in a digon of D. Thus it corresponds to an arc (u, v) in D, where u and v are non-adjacent nodes in T. Then, clearly, adding (u, v) to the path from v to u in T gives a cycle of length at least 3 in D.

Suppose $\mathcal{D}$ contains a simple cycle $\mathcal{C} = \{T_1, T_2, \ldots, T_r, T_1\}$ where $r \geq 2$ and the T_i's are digon-tree nodes. Then $\mathcal{C}$ easily extends to a cycle C in D by following the appropriate directed path within each T_i. This cycle C has length at least 3. If not, then $r = 2$ and there must be a digon in D between T_1 and T_2. But this implies that T_1 and T_2 belong to the same component of the graph induced by digon edges, a contradiction.
($\Leftarrow$) Let $\mathcal{D}$ be an acyclic digon-tree representation of D. Suppose now that D contains a cycle C of length at least 3. If every node of C is contained in the same digon-tree node T of $\mathcal{D}$ then some arc in C, say a, is not part of any digon in T. It follows that a is a loop at node T; hence $\mathcal{D}$ is not acyclic. So C visits nodes in at least two digon-tree nodes, say in the order $\{T_1, T_2, \ldots, T_r, T_1\}$. Thus we have a circuit in $\mathcal{D}$, and so, again, the digon-tree representation cannot be acyclic. □

COROLLARY 3.3. *G contains no sawtooth cycles if and only if its auxiliary graph D has an acyclic digon-tree representation $\mathcal{D}$.*

PROOF. By Theorem 3.1, G contains no sawtooth cycles if and only if every cycle in D is a digon. By Theorem 3.2, this arises if and only if D has an acyclic digon-tree representation. □

Note that the proof of Theorem 3.2 gives a polynomial time method to find acyclic digon-tree representations and sawtooth cycles.

3.3 Induced Trees

We now investigate what acyclic digon-tree representations tell us about the structure of the support for the current flow f.

THEOREM 3.4. *Let G contain no sawtooth cycles. Then G is the union of edges in a set $\mathcal{T}$ of node-induced trees in G such that any node v is in at most two of the trees and moreover*
(i) All outgoing arcs from v are contained in the same tree.
(ii) All incoming arcs at v are contained in the same tree.

PROOF. Given a graph G with no sawtooth cycles, take the auxiliary graph D of Section 3.2. By Corollary 3.3, we know that D has an acyclic digon-tree representation $\mathcal{D}$.

Each digon in D corresponds to an arc in G. So each digon-tree corresponds to an underlying tree (not necessarily induced) in G. For any node $v \in G$, the nodes v^- and v^+ are contained in different digon-trees in $\mathcal{D}$. If not, we would have a self loop at the digon-tree node of $\mathcal{D}$ containing v^- and v^+. Thus each node indeed lies in at most two trees corresponding to these two digon-tree nodes. Let us now see that each such digon-tree T_1 say actually gives an induced subtree T in G. Suppose to the contrary that there is a pair of nodes $u, v \in V(T)$ such that there is an arc (u, v) in $A(G) - A(T)$. Note that the digon in D between u^- and v^+ does not lie in T_1 by assumption, and hence u^- cannot lie in T_1. By the same reasoning v^+ could not lie in T_1. Hence

T_1 contains u^+, v^- and u^- and v^+ are both in some other digon-tree $T_2 \neq T_1$. But then the two node arcs u^-u^+ and v^-v^+ form a cycle in $\mathcal{D}$ between the nodes representing T_1 and T_2 contradicting the fact that $\mathcal{D}$ is acyclic.

Take a node v with outdegree at least one in G. To see (i), simply note that if (v, w) is an arc in G, then v^- and w^+ are in the same digon-tree node. This implies that all the outgoing arcs from v are contained in the same induced tree. We obtain (ii) in a similar fashion. □

We have from Theorem 3.4, we immediately have

Corollary 3.5. *For each sink node $t_1, t_2, \ldots, t_k$ in G we have that t_i^- is in a singleton digon-tree node. For each source node (zero indegree) $s_1, s_2, \ldots, s_l$ in G we have that s_i^+ is in a singleton digon-tree node.* □

In particular, take a non-singleton digon-tree node $T \in \mathcal{D}$. It contains a set of nodes $X \subseteq \ominus$ and a set of nodes $Y \subseteq \oplus$. By Theorem 3.4, we note the following

Observation 3.6. *The node sets X and Y correspond respectively to disjoint nodes sets X', Y' in G.*

Observation 3.7. *The out-neighbourhood of X' in G, namely $\Gamma_G^+(X')$, is exactly Y'.*

Observation 3.8. *The in-neighbourhood of Y' in G, namely $\Gamma_G^-(Y')$, is exactly X'.*

The structure of the digon-trees is important to us in applying the second phase, where they are used to determine the order in which nodes are processed by the algorithm.

4. THE ALGORITHM: PHASE II - FLOW D-FURCATION

The second phase starts from the graph G and processes it based on the digon-tree representation of its auxiliary graph. This is done in *rounds*. Each round is identified with a digon-tree node T^* in the associated auxiliary digraph $\mathcal{D}$. Within each round, we make *steps* as we process the source nodes s of G (that is, the remaining part of G that we are still processing) that lie in this digon-tree one by one. Processing s means that we determine how to redirect its flow to at most d of its out-neighbours, after which we delete s from the graph.

Note that in Phase II, we keep track of the *old flow* values $f(u, v)$ but we also have to add some *new flow* whenever we redirect our flow. We refer to the *total flow* at a node v to be the sum of the old flow $f(v) = \sum_{(u,v)\in G} f(u, v)$ plus whatever new (shunted) flow has found its way to v. Our goal is to bound this total flow at each node. We now give the details of how we pick our nodes to process, and how to redirect flow.

We process the digon-tree nodes in the reverse of the acyclic ordering defined in Section 3.2. That is, at each step we pick a new digon-tree node T^* such that there are no arcs leaving T^* in $\mathcal{D}$. Note that if s is some source node in G, then by Corollary 3.5, s^+ would be a singleton-digon-tree node with a single arc entering it. We assume that such singleton nodes are always removed first. If this is the case, then the next digon-tree node T^* in the acyclic order is not be a singleton node, and since it has no arcs leaving it, its $\ominus$ nodes correspond to sources in the current subgraph of G. This is summarized as:

Lemma 4.1. *If G still contains some arcs, then in the auxiliary digraph D, there is a non-singleton digon-tree node T^* whose set of $\ominus$ nodes corresponds to a subset of sources in G.* □

We search for s, the next node to be processed, amongst the nodes in the digon-tree node T^* specified by Lemma 4.1.

Lemma 4.2. *Let T be a tree with bipartition classes X and Y such that each node of X has degree at least 2. Then T contains a node $s \in X$ with at most one non-leaf neighbour in T.*

Proof. Let $Y_I \subseteq Y$ be the set of non-leaf nodes in Y. Now suppose every node in X has at least two neighbours in Y_I. Clearly, each node $b \in Y_I$ has all its neighbours in X; there are at least two such neighbours as b is a non-leaf. Therefore, the nodes $X \cup Y_I$ induce a bipartite graph with minimum degree 2. It follows that T contains a cycle, a contradiction. □

Let T^* be our next non-singleton digon tree node. We may apply Lemma 4.2 to its underlying undirected graph; otherwise there is a node in G with outdegree one and we would have performed a contraction. Ergo, there is a source node $s \in G$ for which s^- has at most one non-leaf neighbour in T^*. We process this node in our next step. The existence of such a source node is the key for our approach.

We now complete the description by describing how flow is redirected. We denote by U the original maximum load on any node. Our goal is to show that we can create a d-furcated flow by processing nodes (in the order described above) such that for each node v, its total load is at most $f(v) + \frac{1}{d-1}U$. We prove this inductively, with the additional constraint that at each stage any non-source node has load still equal to $f(v)$, its original load. This is because we have yet to redirect any new flow to such nodes. Note that by Observation 3.8, at the end of any round of Phase II, the plus nodes in T^* now correspond to sources in the remaining graph G. Phase II terminates when G consists only of the sink nodes $\{t_1, t_2, \ldots, t_k\}$ at which point all the nodes will be overloaded by at most U.

Theorem 4.3. *There is a d-furcated flow such that each node v has load at most $f(v) + \frac{1}{d-1}U$. In particular, there is a d-furcated flow with node congestion at most $(1 + \frac{1}{d-1})U$.*

Proof. Consider $s^- \in T^*$. By induction, s is currently overloaded by at most $\frac{1}{d-1}U$. Now all of the neighbours of s^- in T^* bar at most one, are leaves. Let the out-neighbours of s in T^* (and hence the remaining part of G by Observation 3.7) be $u_1, u_2, \ldots, u_r$. Letting $u_1, u_2, \ldots, u_{r-1}$ be leaves in the digon-tree node, we then have two cases.

(i) $r \leq d$: Keep the flows $f(s, u_1), f(s, u_2), \ldots, f(s, u_r)$ the same, but move the (possibly) additional $\frac{1}{d-1}U$ load from s to u_1, say. Since u_1 is a leaf, by Observation 3.8, it becomes a source after the removal of s with load at most $f(u_1) + \frac{1}{d-1}U$.

(ii) $r \geq d + 1$: so $u_1, u_2, \ldots, u_d$ are leaves. Clearly

$$\sum_{j>d} f(s, u_j) \leq f(s) \leq U$$

We remove the arcs $\{(s, u_j) : j > d\}$ and split this flow equally on the arcs $(s, u_1), \ldots, (s, u_d)$. In addition, since

s is a source it may have an additional bundle of demand of size $\frac{1}{d-1}U$. This bundle is also split equally on the arcs $(s, u_1), \ldots, (s, u_d)$. Thus the total load at u_1 (resp. $u_2, \ldots, u_d$) increases by at most

$$\begin{aligned}\frac{1}{d}\left(\sum_{j>d} f(s, u_j) + \frac{1}{d-1}U\right) &\leq \frac{1}{d}(f(s) + \frac{1}{d-1}U) \\ &\leq \frac{1}{d}\left((1 + \frac{1}{d-1})U\right) \\ &= \frac{1}{d-1}U\end{aligned}$$

Since $u_1, \ldots, u_d$ are leaves, by Observation 3.8, they all become sources after the removal of s. The result now follows. $\square$

Since we have assumed that $U = 1$, we see that if G has a fractional flow with maximum load one then it has a d-furcated flow with maximum load $1 + \frac{1}{d-1}$.

5. A MATCHING LOWER BOUND

We now present an example to show that this upper bound on the congestion gap is tight.

THEOREM 5.1. *For all $\epsilon > 0$ there is a network having a flow with maximum node load 1 but where all d-furcated flows have node congestion at least $1 + \frac{1}{d-1} - \epsilon$.*

PROOF. We construct a family of networks $N(k, m)$ with optimal fractional flows of maximum load one. However, for all $\epsilon > 0$, there is a member of this family for which any d-furcated flow must have congestion at least $1 + \frac{1}{d-1} - \epsilon$. The network $N(k, m)$ is just a k-ary tree of depth m; thus, there are k^i nodes at level i. The root node has demand one and all other nodes have demand $(k-1)/k$. This is illustrated in Figure 2.

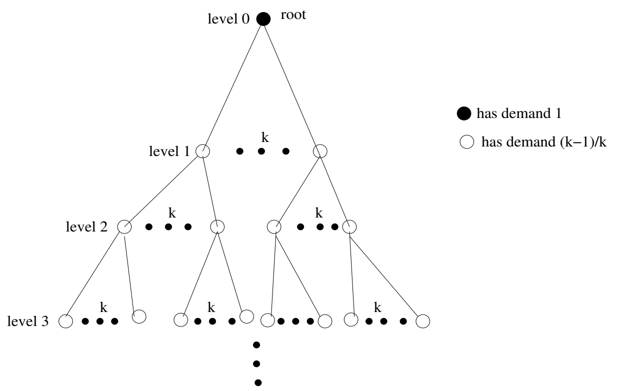

Figure 2: Lower bound construction.

Clearly, the flow where each node sends $1/k$ flow to each of its neighbours at the next level has maximum load equal to 1. We show by induction on level number that at each level i in $N(k, m)$, a d-furcated flow has some node with congestion at least $C_i(k)$ where

$$C_i(k) = \frac{1}{d^i} + \frac{1}{d-1}\frac{k-1}{k}\left(d - \frac{1}{d^{i-1}}\right)$$

This is true for level $i = 1$ since some node has congestion at least

$$\frac{1}{d} + \frac{k-1}{k} = C_1(k)$$

Now suppose there is a node v_i at level i with load at least $C_i(k)$. Then some neighbour v_{i+1} at level $i+1$ receives at least a $1/d$ fraction of this in the d-furcated flow. Thus the congestion C at v_{i+1} is at least $C_i(k)/d + (k-1)/k$. That is,

$$\begin{aligned}C &\geq \frac{1}{d}\left(\frac{1}{d^i} + \frac{1}{d-1}\frac{k-1}{k}\left(d - \frac{1}{d^{i-1}}\right)\right) + \frac{(k-1)}{k} \\ &= \frac{1}{d^{i+1}} + \frac{k-1}{k}\left(1 + \frac{1}{d}\left(\frac{d}{d-1} - \frac{1}{(d-1)d^{i-1}}\right)\right) \\ &= \frac{1}{d^{i+1}} + \frac{k-1}{k}\left(1 + \frac{1}{d-1} - \frac{1}{(d-1)d^i}\right) \\ &= \frac{1}{d^{i+1}} + \frac{1}{d-1}\frac{k-1}{k}\left(d - \frac{1}{d^i}\right) \\ &= C_{i+1}(k)\end{aligned}$$

Thus the congestion of $N(k, m)$ is at least $C_m(k)$ and for every $\epsilon > 0$, clearly there are sufficiently large values of m and k so that $C_m(k) \geq 1 + \frac{1}{d-1} - \epsilon$. $\square$

6. A HARDNESS RESULT

The proof of our congestion bound gives a polynomial time algorithm to convert a fractional flow into a d-furcated flow with congestion at most $1 + \frac{1}{d-1}$. For example, we have a factor 2-approximation algorithm for the problem of finding a minimum congestion bifurcated flow, denoted BIFURCATED-FLOW. In this section we show that this problem is maxSNP-hard. (Similarly, it may be shown that the problem of finding a minimum congestion d-furcated flow is maxSNP-hard for fixed d.)

To prove this we give a reduction from 3SAT. Moreover, by applying standard transformations, we may assume that each literal appears in exactly three clauses and that each clause contains exactly three literals. We take such a 3SAT instance and reduce it to a bifurcated flow problem as follows.

We have a *gadget* for each variable x_i. This gadget consists of just four nodes. There is a *selector node* s_i with outgoing arcs to two *literal nodes*, namely x_i and $\bar{x}_i$, and to a dummy (variable) sink d_i. Our construction ensures that s_i must send flow to d_i in any low congestion solution. Therefore, in a bifurcated flow, the other outgoing arc from s_i can be viewed as a truth assignment for the variable x_i. Specifically, if s_i sends flow to x_i then x_i is set to be true; if s_i sends flow to $\bar{x}_i$ then x_i is set to be false.

In addition, x_i will be the root of a binary tree with three leaves. Each leaf is connected to a *clause sink* C_j where C_j is one of the three clauses containing the literal $\bar{x}_i$. The leaf is also connected to its own dummy sink. Similarly, $\bar{x}_i$ is the root of a binary tree whose leaves are each connected to a clause that contains the literal x_i, as well as to a dummy sink. This construction is illustrated in Figure 3. For motivation, sending flow through a tree results in setting the corresponding literal to true. If all three binary trees that a clause sink is connected to are true, then the clause itself would not be satisfied (as it connects to trees rooted at the negations of the literals it contains).

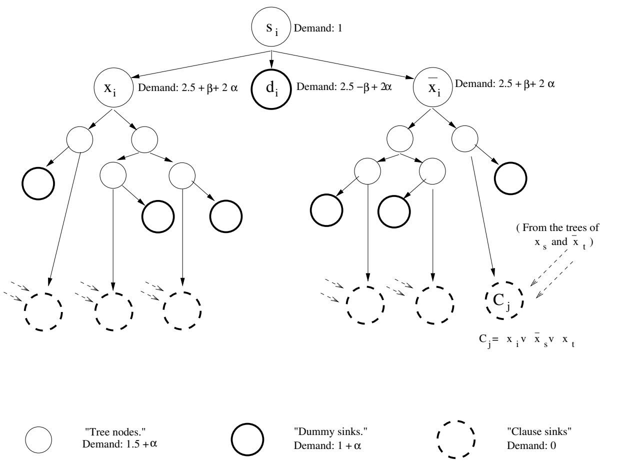

Figure 3: Hardness Gadgets.

It remains to specify the demand of each node. For the variable gadget: each selector node has demand 1, each literal node has demand $2.5+\beta+2\alpha$, and the dummy variable sink d_i has demand $2.5-\beta+2\alpha$. Each node in the binary tree associated with a literal has demand $1.5+\alpha$, the other dummy sinks have demand $1+\alpha$. Finally, each clause sink has demand 0.

THEOREM 6.1. BIFURCATED-FLOW *is maxSNP-hard.*

PROOF. First we show that there is a bifurcated flow of congestion $3+2\alpha$ if there is a satisfying assignment. If x_i is true in this assignment, then send $0.5-\beta$ units of flow from s_i to x_i; in this case, we say that the binary tree for x_i is *selected*. Otherwise send $0.5-\beta$ units to $\bar{x}_i$ and say that its binary tree is selected. In either case send $0.5+\beta$ units to d_i. Thus the loads of d_i and the root of the selected tree are both $3+2\alpha$. The load of the root of the non-selected tree is $2.5+\beta+2\alpha$.

Now if every node in a selected tree splits its load evenly, then each node in the tree also has congestion $3+2\alpha$. From the variable-clause leafs we may then send $2+\alpha$ units to their dummy sinks and $1+\alpha$ to the clause sink. The total demand in a non-selected tree is $8\frac{1}{2}+\beta+6\alpha$. This can be directed so that each leaf has load $2\frac{5}{6}+\frac{1}{3}\beta+2\alpha = 3+\alpha$ if we set $\alpha = \frac{1}{6}(1-2\beta)$. These leafs can then send $2+\alpha$ units to their dummy sinks and 1 to the clause sink. Since we have a satisfying assignment, any clause sink receives flow from an unselected tree (recall that a tree rooted at a literal x_i is connected to clauses containing its negation). Therefore the congestion at a clause sink is at most $3+2\alpha$.

We now show that any bifurcated flow f with congestion $3+2\alpha+\delta$, for a suitable $0<\delta<\beta$, must correspond to a satisfying assignment. Observe that f must send flow to each dummy variable sink. Otherwise either x_i or $\bar{x}_i$ has congestion $3+2\alpha+\beta > 3+2\alpha+\delta$. Moreover, we may assume that f sends $\frac{1}{2}+\beta+\delta$ flow to d_i as this does not increase the congestion of the flow. So such a flow corresponds to a truth assignment. We need to show that this must be a satisfying assignment. Suppose not, then there is some unsatisfied clause C_j. The clause sink C_j has congestion at most $3+2\alpha+\delta$, so one of its incoming arcs (u, C_j) must contribute at most $1+\frac{2}{3}\alpha+\frac{1}{3}\delta$. As u is a leaf in one of the binary trees, it may also send $2+\alpha+\delta$ flow to its dummy node. Moreover, since C_j is not satisfied, we know that u is in a selected tree. Therefore, one of other two leaves in that tree has load at least $\frac{1}{2}\big((9+6\alpha-\delta)-(1+\frac{2}{3}\alpha+\frac{1}{3}\delta)-(2+\alpha+\delta)\big) = 3+\frac{13}{6}\alpha-\frac{7}{6}\delta$. This implies that $\frac{1}{6}\alpha \le \frac{13}{6}\delta$, that is $\delta \ge \frac{1}{13}\alpha = \frac{1}{78}(1-2\beta)$. Setting $\beta=\frac{1}{80}$, we see that we cannot get a better approximation guarantee for BIFURCATED-FLOW than $1+\frac{1}{266} \approx 1.0038$ unless P=NP. □

7. UNSPLITTABLE FLOWS

Suppose we make an additional restriction on our network flows. What if, in addition to flows being d-furcated, individual demands must be routed *unsplittably*? That is, each demand r_v must be routed along a single path. We let d_{max} be the maximum demand amongst our various commodities. (Note that for unsplittable flow, one may have multiple demands associated with a node V; for our discussion, we may amalgamate them into one large demand at v without affecting the results.) Combining our approach with the unsplittable flow techniques of Dinitz et al. [4] we obtain the following result.

THEOREM 7.1. *Given a fractional flow of with maximum node load U, there is a d-furcated, unsplittable flow with congestion at most $(1+\frac{1}{d-1})U + d_{max}$.*

PROOF. To see this, first find a d-furcated flow with congestion $(1+\frac{1}{d-1})$ as described. Then run the unsplittable flow algorithm [4] on this d-furcated flow. The resultant congestion increases by at most the maximum demand. □

Incorporating the approach of Dinitz et al. gives us approximation algorithms for other objective functions. For instance, solving the *maximum routable demand problem* where we wish to unsplittably route a subset of the demands, of maximum total weight, using a d-furcated flow of congestion U. One can also obtain results for the problem of routing all demands in a minimum number of *rounds*. We are required to route all demands unsplittably (obeying some maximum node load constraint) using d-furcated flows. We defer the details to the full version of the paper.

8. CONCLUSION

We have shown that one may produce a d-furcated flow from a fractional flow without increasing the maximum node load by more than a factor of $1+\frac{1}{d-1}$; we have also seen this is tight. Many interesting problems remain. What about multicommodity flow problems with multiple sinks? What happens in networks with non-uniform capacities or costs?

It is natural to ask about a special class of d-furcated flows where each node must split its flow equally along its outgoing arcs (as discussed, such flows are also of practical interest). For example, in networks with maximum outdegree two we call such flows *halfluent*. One may show that the congestion gap between bifurcated flows and halfluent flows may be as large as 2. It may be possible to extend the techniques developed here to obtain an $O(1)$ congestion gap for halfluent flows.

Acknowledgments. We are grateful to R. Kleinberg for a stimulating conversation. P. Donovan was supported in part by NSERC grant 28833 − 04. B. Shepherd and G. Wilfong have been partly supported by basic research grant number N000140610396 from the ONR to Bell Laboratories. A. Vetta is supported in part by NSERC grant 28833 − 04 and FQRNT grant NC−98649.

9. REFERENCES

[1] R. Ahuja, T. Magnanti and J. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, 1993.

[2] J. Chen, R. Kleinberg, L. Lovasz, R. Rajaraman, R. Sundaram and A. Vetta, "(Almost) tight bounds and existence theorems for confluent flows", *Proceedings of the 36th ACM Symposium on Theory of Computing (STOC)*, pp529-538, 2004.

[3] J. Chen, R. Rajaraman, and R. Sundaram, "Meet and merge: approximation algorithms for confluent flow", *Proceedings of the 35th ACM Symposium on Theory of Computing (STOC)*, pp373-382, 2003.

[4] Y. Dinitz, N. Garg and M. Goemans, "On the single-source unsplittable flow problem", *Combinatorica*, **19**, pp17-41, 1999.

[5] J. Kleinberg, "Single-source unsplittable flow", *Proceedings of the 37th on Foundations of Computer Science (FOCS)*, pp68-77, 1996.

[6] S. Kolliopoulos and C. Stein, "Improved approximation algorithms for unsplittable flow problems", *Proceedings of the 38th on Foundations of Computer Science (FOCS)*, pp426-435, 1997.

[7] J. Fong, A. C. Gilbert, S. Kannan, and M. Strauss, "Better alternatives to OSPF routing", *Algorithmica* (Special issue on network design), **43(1-2)**, pp113-131, 2005

[8] B. Fortz and M. Thorup, "Optimizing OSPF/IS-IS weights in a changing world", *IEEE Journal on Selected Areas in Communications* (Special Issue on Recent Advances on Fundamentals of Network Management), **20(4)**, pp756-767, 2002.

[9] B. Shepherd and A. Vetta, "Visualizing, finding and packing dijoins", in D. Avis, A. Hertz, O. Marcotte (eds.), *Graph Theory and Combinatorial Optimization*, Kluwer, pp219-254, 2005.