Unsplittable Flows

Hoon Cho (hhcho@mit.edu) and Alex Wein (awein@mit.edu)

6.854 Final Project

Abstract

An unsplittable flow in a multicommodity network is a flow that routes each commoidty on a single path. In this paper, we introduce the unsplittable flow problem and an open conjecture of Goemans, which states that given a fractional flow it is possible to construct an unsplittable flow that only overflows each edge by at most the maximum demand of a commodity. We provide an intuitive explanation of the proofs of the conjecture for the following two special cases: unweighted graphs [1], and weighted 2-tier graphs for which the unsplittable flow problem is equivalent to a particular scheduling problem [2]. Then, we describe our unsuccessful attempt at extending the proof of the conjecture for 2-tier graphs to more general 3-tier graphs and discuss the difficulties and limitations of our approach.

1 Introduction

We will be discussing a variant of max flow called the single-source unsplittable flow problem. Suppose we are given a directed graph $G = (V, E)$ with a single source s and multiple terminals $t_1, \ldots, t_k$ with corresponding demands $d_1, \ldots, d_k > 0$. Each terminal is located at some node in the graph and we allow multiple terminals to share a single node. A flow is said to meet the demands d_i if the net inflow at every node (except the source) is exactly equal to the sum of the demands of terminals located at that node. In particular, the flow must be conservative at all nodes (except the source) that do not contain terminals.

If we impose capacities on the edges and ask whether there exists a flow meeting all the demands, we have arrived at the single-source multicommodity flow problem. This problem captures the following real-world shipping problem. Suppose we have a number of different commodities, say apples, bananas and oranges. The graph represents a network of possible shipping routes. We want to deliver d_1 units of apples from s to t_1, d_2 units of bananas from s to t_2, and d_3 units of oranges from s to t_3. The total number of units (apples plus bananas plus oranges) travelling along a particular shipping route cannot exceed the capacity on that edge. The multicommodity flow problem asks whether it is possible to simultaneously satisfy all the demands without exceeding any of the capacities.

Note that the multicommodity flow problem can be reduced to the max flow problem as follows. Given an instance of multicommodity flow, add a supersink T and for each i, an edge (t_i, T) with capacity d_i. Then check whether the resulting max flow instance has a flow that saturates all edges (t_i, T).

It is important to recognize the following subtlety. We defined a solution to multicommodity flow to simply be a flow $f : E \to [0, \infty)$ on the graph. This flow tells us the total number of units shipped along a particular route, but it does not tell us how many of these units are apples, how many are bananas, and how many are oranges. A more informative solution would be what we are going to call a *flow with allocations*. A flow with allocations is a flow f that solves multicommodity flow together with a decomposition $f = f_1 + \cdots + f_k$ of f into one flow for each commodity. The flow f_i must have outflow d_i at s, inflow d_i at t_i, and must be conservative at all other nodes. A flow with allocations tells us how the individual commodities are routed. Above we have shown how to find a flow (without allocations) for the multicommodity flow problem (provided that one exists), but does this mean we can find allocations for that flow? The answer is yes. To do this, decompose the flow into paths and cycles. Remove all the cycles and note that the resulting flow still satisfies all the demands. Then consider the terminals t_i one by one and assign to each t_i enough paths (or

fractions of paths) from s to t_i to exactly meet the demand d_i. This justifies asking only for a flow (without allocations) as a solution to multicommodity flow.

Now we are ready to introduce the concept of unsplittable flows. So far we have considered flows that may be *fractional* in the following sense. In our fruit-delivery problem, the solution might take half the apples along one path to t_1 and the other half along a different path to t_1. But what if we want to send all the apples along a single path, send all the bananas along a single (possibly different) path, and so on? This is the idea behind unsplittable flows. A flow with allocations $f = f_1 + \cdots + f_k$ (as defined above) is *unsplittable* if for each i there exists a path from s to t_i such that f_i is zero everywhere except on that path. To emphasize this distinction, arbitrary flows that are not necessarily unsplittable will be called *fractional*. Note that an unsplittable flow is, by definition, "with allocations". On the other hand, fractional flows are assumed to be "without allocations" unless otherwise specified.

Now that we have introduced the notions of fractional flows and unsplittable flows, we might wonder how restrictive the unsplittable condition is. For instance, if we have a fractional flow that meets particular demands, can we produce an unsplittable flow that meets those same demands without increasing the amount of flow on any edge? It is easy to see that the answer is no. However, an algorithm by Dinitz, Garg, and Goemans [1] shows that a fractional flow can be converted to an unsplittable flow without increasing the flow on each edge by "too much" in the following sense. Let $d_{max} = \max_i d_i$ be the largest demand.

Theorem 1. *(Dinitz, Garg, Goemans [1]) Given a fractional flow f meeting demands d_i, there exists an unsplittable flow meeting the same demands such that the total flow on any edge e is at most $f(e) + d_{max}$.*

In converting a fractional flow to an unsplittable flow, you only need to increase the flow on each edge by at most d_{max}. We cannot hope to do better than this because [1] gives examples of instances for which this increase is unavoidable.

A natural extension of the unsplittable flows problem is the weighted version in which a nonnegative cost $c(e)$ is assigned to each edge e. The total cost of a flow is $\sum_{e \in E} f(e)c(e)$. Now the objective is to convert a fractional flow to an unsplittable flow as above, with the additional constraint that the total cost may not increase. As discussed in [3], Goemans has conjectured that this is possible.

Conjecture 1. *(Goemans) Given a fractional flow f of total cost C that meets demands d_i, there exists an unsplittable flow with cost $\leq C$ meeting the same demands such that the total flow on any edge e is at most $f(e) + d_{max}$.*

We will start by explaining the proofs of two special cases of this conjecture. The first is the unweighted case, Theorem 1, where the costs are all zero. The second is the weighted case on a restricted class of graphs corresponding to a particular well-studied scheduling problem. We will then explain some of our attempts to prove the conjecture and why they failed.

2 Proof for the Unweighted Case

In this section we present the proof of Theorem 1 above, the unweighted (i.e., no costs) version of the Goemans conjecture. The proof is an algorithm that takes an initial fractional flow and produces the desired unsplittable flow. In the following sections we describe this algorithm, explain the intuition behind it, and prove its correctness. We start with an overview of the main features of the algorithm before giving a complete description.

2.1 Movement of Terminals

The algorithm makes progress by moving terminals closer to the source in order to reduce the problem to a smaller subproblem. Suppose terminal t_i is located at vertex v and the algorithm has just decided that the unsplittable path for t_i should arrive at v using edge $e = (u, v)$. (This of course requires $f(e) \geq d_i$.) Then we move terminal t_i from v to u and decrease the flow on edge e by d_i units. This gives us a smaller subproblem.

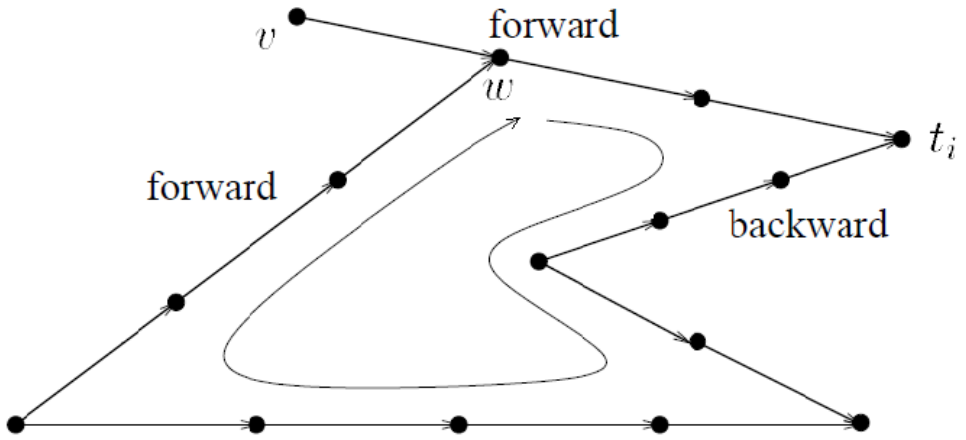
Once we solve the smaller problem we can add the d_i units back to e to get our final solution. The algorithm will move terminals closer to the source until all terminals arrive at s, at which point the algorithm is done. The very first step of the algorithm will take the input fractional flow and make it acyclic (by applying flow decomposition and removing cycles) so that the notion of "closer to the source" is well-defined. Note that when we move terminals we decrease the flow on certain edges. If the flow on an edge reaches zero we remove the edge from the graph in the smaller subproblem we have just created. This means we can assume at all times that every edge has positive flow, an assumption that will be necessary in the next section. Note that to extract the final solution we can take a terminal and retrace the path through which it moved throughout the execution of the algorithm. This path is the path that should be used to route demand to that terminal. Equivalently, the demands routed across a given edge precisely correspond to the terminals that moved across that edge at some point during the execution of the algorithm.

2.2 Alternating Cycles

The fundamental step in many flow algorithms is to find some sort of "augmenting structure" in the graph and to "augment" flow along it in some way. This algorithm is no exception. In particular, the augmenting structure is something called an *alternating cycle*, which can be found using the following simple steps.

1. Start at any vertex v.
2. Never go back on the same edge that you just arrived on.
3. Follow a forward edge (e.g., follow $e = (u, v)$ from u to v) whenever possible.
4. Follow a backward edge (e.g., follow $e = (u, v)$ from v to u) if you cannot follow a forward edge without violating step 2.
5. When you reach a vertex w that you have visited before, stop. The resulting closed loop (without the path from v to w) is an alternating cycle.

The following figure from [1] illustrates an alternating cycle.



It is called alternating because of its alternating forward and backward paths. Note that if a vertex has only one incident edge (including both incoming and outgoing edges) then the procedure above might get stuck and fail to find an alternating cycle. In Section 2.3 we will show that the algorithm ensures that this never happens. Note that a vertex at which you change from going forward to going backward must contain at least one terminal. This is because the only time you change from going forward to going backward is if you arrive at a node with no outgoing edges. But such a node must have at least one terminal because a vertex without any terminals must obey flow conservation and must therefore have at least one outgoing edge.

Once we have an alternating cycle we can augment flow along it as follows. Decrease flow on the forward edges and simultaneously increase flow on the backward edges by the same amount until one of the following happens:

- The flow on some (forward) edge drops to zero. In this case the edge is removed from the graph, thereby making progress.
- For some terminal t_j at some vertex x and for some (backward) incoming edge (y, x), $f(y, x)$ increases from below d_j to d_j. This makes progress because it will allow us to move terminal t_j from x to y.

Note that augmenting along an alternating cycle preserves flow conservation. However, it does increase the flow on some edges (namely the backward edges) which is a potential problem because we need to make sure the flow on each edge increases by no more than d_{max}. We will explain how this is resolved in Section 2.3.

2.3 The Algorithm

Now we are ready to give a complete description of the algorithm. We will attempt to present it in an intuitive way in which all definitions and rules are motivated. The essence of the algorithm is the two steps described above. In each iteration we find an alternating cycle and augment flow along it. Then we move terminals closer to the source according to a certain rule. However, there are some technical details that make the algorithm somewhat more complicated. These are to deal with two important issues:

(i) We need to make sure that the flow on each edge does not increase by more than d_{max}.

(ii) We need to ensure that we can always find an alternating cycle.

We will start by talking about how to resolve issue (i). First we need a definition.

Definition 1. *An edge (u, v) is* singular *if v and all the vertices reachable from v have out-degree at most 1. Equivalently, the vertices reachable from v induce a directed path.*

The motivation behind this definition is the following. Note that every backward edge in an alternating cycle is singular. Furthermore, recall that the only edges to which flow is added are the backward edges in an alternating cycle. Also note that once an edge becomes singular it can never go back because edges can only be removed from the graph and never created. The result of all this is that the algorithm only adds flow to an edge after it becomes singular. Recall that for a given edge e, the demands routed across it are those corresponding to the terminals that move across it at some point during the execution of the algorithm. Since we only add flow to singular edges, we now know that all the terminals that move across e before e becomes singular have total demand at most $f(e)$ where f is the initial fractional flow. This is because when we move a terminal t_i across an edge e, we decrease $f(e)$ by d_i and $f(e)$ remains nonnegative.

As a result, we can resolve issue (i) by ensuring that at most 1 terminal moves across e after it becomes singular. We can do this by making sure that whenever a terminal moves across a singular edge, the singular edge is left with zero flow and is removed from the graph. This motivates the following rules for moving terminals at each iteration of the algorithm: move terminal t_i from v to u whenever either

(1) (u, v) is a singular edge and $f(u, v) = d_i$, or

(2) (u, v) is not singular and $f(u, v) \geq d_i$.

If it is possible to do either (1) or (2), choose (1). These rules essentially say "move terminals whenever possible, but don't move across a singular edge unless it causes the edge to be removed." During each terminal-moving phase we move terminals according to rules (1) and (2) until no more terminals can be moved. This means a single terminal might move across more than one edge.

Finally we have motivated all the steps in the algorithm. We still need to resolve issue (ii) but first we will formally describe the algorithm.

Algorithm 1. (Dinitz, Garg, Goemans [1])

Input: $G = (V, E)$ with source s and terminals $t_1, \ldots, t_k$, demands $d_1, \ldots, d_k$, fractional flow f

1. Use flow decomposition to make f acyclic.
2. Move terminals closer to the source wherever possible.*
3. Identify singular edges and label them as singular.
4. Find an alternating cycle and augment flow along it.
5. Move terminals according to rules (1) and (2) above, using the labels from step 3.**
6. If all terminals have reached the source, stop. Otherwise go to step 3.

* Step 2 does not adhere to rules (1) and (2) above. It simply moves t_i from v to u whenever $f(u, v) \geq d_i$. This does not interfere with our resolution of issue (i) because flow has not been increased on any edges yet.

** In step 5, an edge is only considered singular if it was singular at the beginning of the iteration (step 3). We will need this fact later for resolving issue (ii). Note that this does not interfere with our resolution of issue (i) because an edge that just became singular (in step 4) has not had any flow added to it yet. This is clear because we only add flow to backward edges in the alternating cycle, and the alternating cycle is chosen so that all backward edges are already singular.

Now we need to resolve issue (ii) by making sure that the algorithm never gets stuck while trying to find an alternating cycle. No additional modifications to the algorithm are necessary in order to guarantee this. As mentioned earlier, the only way the algorithm can fail to find an alternating cycle is if it arrives at a degree-1 (including both incoming and outgoing edges) vertex via its only incident edge. We will prove that this can never happen.

First consider the case where you reach a vertex with a single outgoing edge (and no incoming edges). Due to flow conservation, this vertex must be the source. If the algorithm chose to start at the source when building an alternating cycle then it is not actually stuck because it can just move forward along the one outgoing edge. The case we have to worry about is if a backward path goes all the way back to the source. But recall that backward edges are singular and so this can only happen if the single outgoing edge from the source is singular (i.e., the entire graph is a single directed path). If this is the case it is easy to see (by induction, starting at the end of the path) that all terminals should already have been moved to the source. This rules out the problem of getting stuck at a vertex with a single outgoing edge.

Now we will rule out the problem of getting stuck at a vertex v with a single incoming edge (u, v) (and no outgoing edges). If we suppose this problem arises then by flow conservation, v must contain at least one terminal t_i. Since (u, v) is the only incoming edge for v, it must have enough flow to meet t_i's demand, i.e. $f(u, v) \geq d_i$. This motivates the following definition.

Definition 2. *A terminal t_i at vertex v is* irregular *if it has an incoming edge (u, v) with $f(u, v) \geq d_i$. Otherwise it is called* regular.

We have just seen that v is an irregular terminal. The following fact about irregular terminals completes our resolution of issue (ii) because it contradicts our assumption that v has a single incoming edge.

Lemma 1. *A vertex containing an irregular terminal must have at least 2 incoming edges.*

To prove this statement, consider the same setup above where irregular terminal t_i is stationed at vertex v with $f(u, v) \geq d_i$. We might wonder why t_i was not moved from v to u last iteration. According to the rules for moving terminals, this can only happen if (u, v) is singular and $f(u, v) > d_i$. Note that before the first iteration of the algorithm (right after step 2), there are no irregular terminals. So how can an irregular terminal be created? The rules for augmenting flow prevent $f(u, v)$ from going from below d_i to above d_i while t_i is at v, so v must have moved to terminal v from say w, and at this point we must already have had $f(u, v) > d_i$. Let j be the iteration at which t_i moved to v. At this point (u, v) must already have been singular or else t_i would have moved to u. Recall that in defining the algorithm we needed to add the technical condition that edges did not count as singular (for the purposes of terminal movement) unless they

were already singular at the beginning of the iteration. This means (u, v) was already singular at the beginning of iteration j. But by the definition of singular, (v, w) must also have been singular at the beginning of iteration j. But this means t_i moved across a singular edge. This can only happen if the singular edge was immediately removed from the graph. What we have shown so far is that an irregular terminal can only be created at a vertex if it moves there and destroys the vertex's only outgoing edge in the process. Therefore a given vertex can only contain at most one irregular terminal. To complete the proof, recall $f(u, v) > d_i$ and that v has no more outgoing edges, so v must contain at least one other terminal. This terminal must be regular. By flow considerations this implies that v has at least 2 incoming edges as desired.

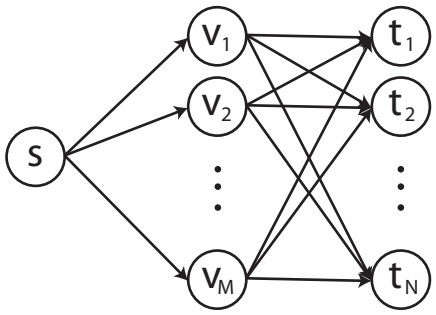
Now we have presented the algorithm and shown that it works, completing the proof of the Goemans conjecture in the unweighted case.

3 Proof of Conjecture on 2-Tier Scheduling Graphs

In this section we describe a result [2] that proves the Goemans conjecture for a particular class of "2-tier" graphs.

Consider the (seemingly-unrelated) problem of scheduling jobs on parallel machines with costs. The objective of the problem is to process all jobs while minimizing the total cost incurred. Scheduling job j on machine i requires a processing time of p_{ij} and incurs a cost of c_{ij}. There is an additional constraint that each job is processed by exactly one machine and that each machine i is only available for T_i time units. Shmoys and Tardos [2] give a polynomial time algorithm that, given a value C, either proves no feasible schedule of cost C exists or finds a schedule of cost at most C where each machine i is used for at most $2T_i$ time units. Interestingly, this result corresponds to a proof of the Goemans conjecture on 2-tier scheduling graphs. In this section, we will outline the proof of the conjecture on 2-tier graphs based on the ideas presented in [2].

First, we establish the connection between the scheduling problem and unsplittable flows. A special case of the scheduling problem where the processing time of each job is constant across all machines ($p_j = p_{ij}, \forall i$) can be viewed as an unsplittable flow problem on the following 2-tier graph



where the terminals $t_1, \ldots, t_N$ represent the jobs and the nodes in the intermediate layer $v_1, \ldots, v_M$ represent the machines. Each terminal t_j has demand p_j. Each edge connecting the source s to v_i has zero cost and capacity T_i to ensure that only a total of T_i processing time is allowed for machine i. In addition, each edge between v_i and t_j has infinite capacity and cost c_{ij}/p_j per unit flow. In this alternative representation, a *flow* of value f along a path $s \to v_i \to t_j$ semantically represents job j being processed for f time units on machine i. Finding a minimum cost *unsplittable flow* meeting all demands is equivalent to solving the original scheduling problem where each job can only be processed by a single machine.

Now, we proceed to describing the proof of the conjecture for this special case. Let us assume we are given a fractional solution that achieves an overall cost C. In particular, let x_{ij} be the fraction of job j allocated to machine i. Then, we have that

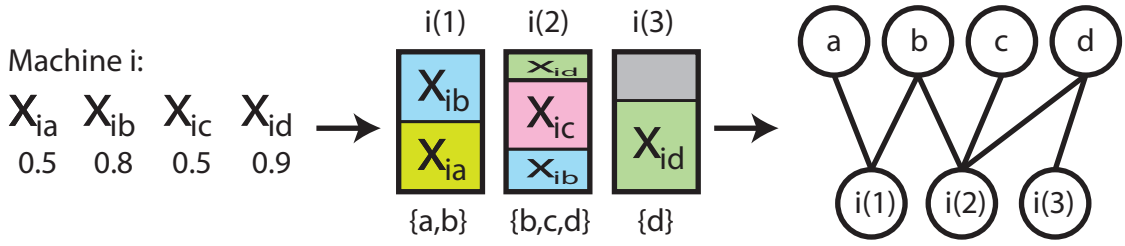
$$\begin{aligned}
\sum_{i,j} c_{ij} x_{ij} &= C, \\
\sum_{j} x_{ij} p_j &\leq T_i, \forall i, \\
\sum_{i} x_{ij} &= 1, \forall j, \\
x_{ij} &\geq 0, \forall i, j.
\end{aligned}$$

The Goemans conjecture posits that we can find an integral solution $\{\tilde{x}_{ij}\}$ (i.e. an unsplittable flow) such that

$$\begin{aligned}
\sum_{i,j} c_{ij} \tilde{x}_{ij} &\leq C, \\
\sum_{j} \tilde{x}_{ij} p_j &\leq T_i + \max_j p_j, \forall i, \\
\sum_{i} \tilde{x}_{ij} &= 1, \forall j, \\
\tilde{x}_{ij} &\in \{0, 1\}, \forall i, j.
\end{aligned}$$

To show this, we are going to use a novel rounding procedure proposed by Shmoys and Tardos [2] based on *binning*. An intuitive explanation of the technique is the following. For each machine, we are given a number of job-pieces (i.e. fractions of jobs) that are scheduled on it. We will pack these job-pieces into small bins in a particular fashion. Each machine will have its own set of bins. At most one job from each bin will be chosen to be scheduled on the machine. The key idea is that the construction guarantees a "cascading" or "shifting" effect where the job chosen from a particular bin has processing time no greater than the total processing time of the previous bin. As a result, the processing time for a machine only increases by $p_{max} = \max_j p_j$ over the fractional solution. The increase in p_{max} is due to the job in the first bin, since there is no previous bin to pair it up with.

Now we give a detailed formal description of the binning idea. For each machine i, we sort the associated positive x_{ij}'s in decreasing order of processing time of job j. Then, we greedily pack these x_{ij}'s into bins of size 1 in that order. If a bin overflows, an x_{ij} may be split between two bins, e.g. we might have $x_{ij} = \frac{2}{3}$ with $\frac{1}{2}$ in one bin and $\frac{1}{6}$ in the next bin. For each machine, all bins except possibly the last will be full, i.e. size 1. Formally, assuming $p_1 \geq \cdots \geq p_N$ without loss of generality, the kth bin for machine i contains job j if $x_{ij} > 0$ and if either $k - 1 \leq \sum_{j'=1}^{j-1} x_{ij'} < k$ or $k - 1 < \sum_{j'=1}^{j} x_{ij'} \leq k$. Note that each job can be in up to two bins for each machine. Once we have created bins for every machine, we construct a bipartite matching graph between jobs and machines where each machine node is split into $\lceil \sum_j x_{ij} \rceil$ nodes, one for each of its bins. Then, we add an edge between job j and the node corresponding to the kth bin of machine i if that bin contains at least part of x_{ij}. The following figure illustrates this process. We are considering a single machine i and jobs a, b, c, and d.



Let us denote the resulting bipartite graph as B. If we set the weight associated with each edge between job j and the kth bin of machine i to be the actual amount of x_{ij} included in the corresponding bin and set the cost to be c_{ij}, the set of edge weights over the entire graph forms a fractional matching of jobs to the duplicated machine nodes in B with total cost C. Note that the sum of weights associated with the edges connected to each job i is equal to $\sum_{i=1}^{M} x_{ij} = 1$.

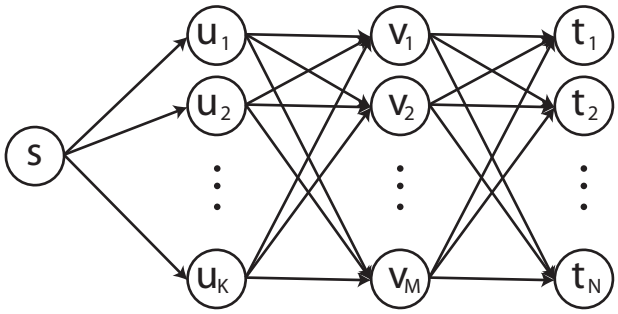
Now, since B is a bipartite graph and there exists a fractional matching that achieves a cost of C, there exists an integral matching with the same cost or less. This is a well-known property that comes from the fact that the polytope for the bipartite matching LP has integral extreme points. Given an integral matching we can construct an unsplittable flow solution by simply assigning job j to machine i if one of the edges between job j and machine i in B is used in the matching. This gives us the *rounded* solution. In order to analyze the amount of workload on each machine in this approximate solution, we consider the workload accounted for by each bin in the original fractional solution. Let b_{ik} be the original workload of the kth bin of machine i, which is defined as the sum of weights associated with edges connected to the bin in B each multiplied by the processing time of the corresponding job. Note $\sum_k b_{ik} = \sum_j p_j x_{ij}$. With the integral matching, each bin is now assigned to at most one job. Because we rearranged the jobs in decreasing order of processing time during the binning procedure, the processing time of job j assigned to the kth bin in the integral matching is at most $b_{i(k-1)}$, since $b_{i(k-1)}$ is a convex combination of processing times that are at least p_j. Thus, the sum of all processing times assigned to machine i is upper bounded by $\max_j p_j + \sum_k b_{ik}$ where the first term bounds the processing time of the job matched with the node corresponding to the first bin. We have shown that the workload on each machine increases by at most $\max_j p_j$ when we round the fractional solution to obtain the unsplittable flow, while the total cost remains at most C. This concludes the proof of the Goemans conjecture on 2-tier scheduling graphs.

4 Our Proof Attempts for the General Conjecture

In this section we discuss our own attempts to prove the Goemans conjecture. None of our attempts were successful but we are able to explain (with proof) why a number of natural approaches to the problem are insufficient to resolve the conjecture.

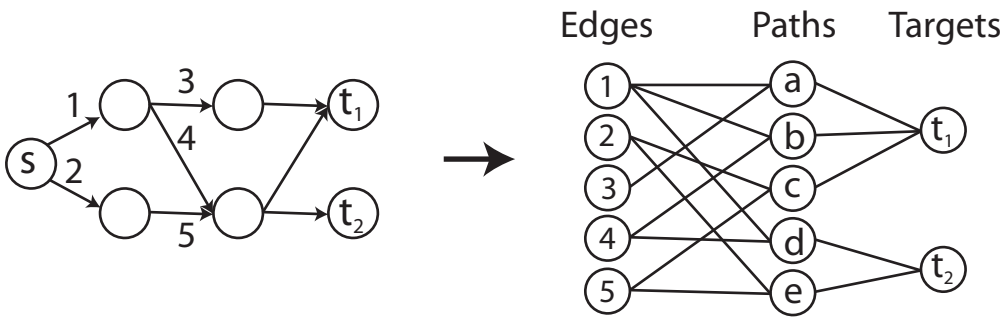
4.1 Extensions of the 2-Tier Scheduling Proof: Matching and Binning

Our initial goal for the project was to extend the result for 2-tier graphs to slightly more complicated 3-tier graphs, which is illustrated below.



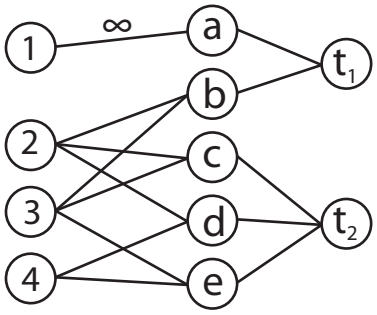
Although we were not able to prove the conjecture for this special case, we obtained some useful insights during the process about what aspects of 3-tier graphs make it harder to prove than 2-tier graphs. We will explain our approach to tackling 3-tier graphs and discuss what we learned in detail.

An important step in the proof of the conjecture for 2-tier scheduling graphs is formulating the rounding procedure as a minimum cost bipartite matching problem, which conveniently has an integral optimal solution. As an attempt to transform 3-tier (or more general) graphs into an equivalent matching problem, we construct the following three-layer *path graph*:



The first layer consists of nodes representing each edge in the original graph except for the ones that are directly connected to the terminals, the second layer consists of nodes representing each unique path from the source to the terminals, and the third layer consists of the terminal nodes. The connectivity between the first and the second layer reflects the set of edges each path contains, and a path node in the second layer is connected to a terminal node in the third layer only if the path leads to that terminal in the original graph. Now an unsplittable flow is just a matching between the terminals and paths that matches every terminal. However, this is not quite a matching problem because we are constrained not by the total demand of terminals assigned to each path node but to the total demand of terminals assigned to each edge node. In order to avoid increasing the flow on any edge by more than d_{max}, the idea is to use a similar binning approach to bin each edge nodes in the first layer of the paths graph based on a fractional solution and to then find an optimal integral matching between the bins and the terminals, mediated by the path nodes. Here, we immediately see the complexity introduced by going from 2-tier to 3-tier graphs. In 2-tier scheduling graphs, each path node is connected to a single node in the first layer, enabling the matching problem to be reduced to bipartite matching by collapsing the intermediate layer. In fact, the resulting bipartite matching graph is equivalent to what we constructed in the proof for 2-tier graphs. However, as soon as we introduce another tier to the graph, selecting a particular path node to match with a terminal forces us to use multiple edge nodes in the first layer. In other words, we have what we call a *fork matching* problem: for each terminal we need to pick a *fork structure* connecting the terminal to a path node and connecting this path node to two edge nodes (or more accurately, to one bin in each of two edge nodes). The costs can be put on the edges in the second (rightmost) tier by computing the total cost of each path. Each fork structure "uses" two edge nodes, and the objective of the fork matching problem is to choose a fork stucture for each terminal so that each edge node is used at most once.

We have now shown that 3-tier unsplittable flows reduces to the fork matching problem in the same way that 2-tier unsplittable flows reduces to bipartite matching. Recall that the 2-tier proof required the fact that every extreme point of the bipartite matching LP is integral. If we could prove that every extreme point of the fork matching LP is also integral then we could complete the proof for the 3-tier case. However, the following counter-example shows that a general fork matching problem may not have an integral optimal (min-cost) solution.



The goal of the fork matching problem in this graph is to match the terminals t_1 and t_2 to the paths a, b, c, d, e. Each chosen path node connects to a bundle of edges, forming a fork structure. As indicated in the figure, assume that the cost on the edge $(1, a)$ is very large. In this particular example, any integral

matching forces t_1 to be matched with 1, because the fork b conflicts with all of the available options for t_2. One can find a better fractional solution by assigning the weight of 1/3 to c, d, and e for t_2, in which case t_1 can assign the weight of 1/3 to b, thereby substantially reducing the overall cost incurred by the edge $(1, a)$. Thus, a minimum cost fork matching problem does not always have an integral solution. It is important to note that this counter-example does not completely undermine the potential of our path graph approach for 3-tier graphs. This is because the graph used in the counter-example is not a path graph generated from an actual multicommodity flow network. This means that we could conceivably still hope that all fork matching problems arising from actual flow networks have integral optimal solutions. However, the counter-example does suggest that we need to consider the structural constraints of realizable path graphs if we want to show integrality of optimal solutions.

4.2 Impossibility Result Regarding Flows With Allocations

In the previous section we explored what we thought was the most natural way to try to extend the proof of the conjecture from the 2-tier case to a more general case. We found that it didn't seem hopeful for a number of reasons, including the fact that the fork matching problem LP does not always have integral extreme points. In this section we prove a more general impossibility result that shows that a particular class of algorithms, which we will call "algorithms with allocated input" cannot be used to prove the conjecture. The approach from the previous section is an algorithm with allocated input, so this section provides evidence that those techniques cannot be applied to fully general flow graphs, at least without significant modifications.

Now we describe what we mean by algorithms with allocated input. Recall from Section 1 that in a k-commodity setting, a flow with allocations is a flow with a decomposition $f = f_1 + \cdots + f_k$ describing how much of each commodity flows on each edge. An algorithm that proves the conjecture would take a flow (without allocations) and produce an unsplittable flow satisfying the conditions of the conjecture. However, consider instead an algorithm that takes as input a flow with allocations $f = f_1 + \cdots + f_k$ and produces an unsplittable flow that *respects* these allocations, i.e. commodity i can only be unsplittably routed across an edge e if $f_i(e) > 0$. Equivalently, suppose our algorithm first takes an input flow and produces arbitrary feasible allocations for it, and then proceeds to find an unsplittable flow that respects these allocations. We call this type of algorithm an *algorithm with allocated input*.

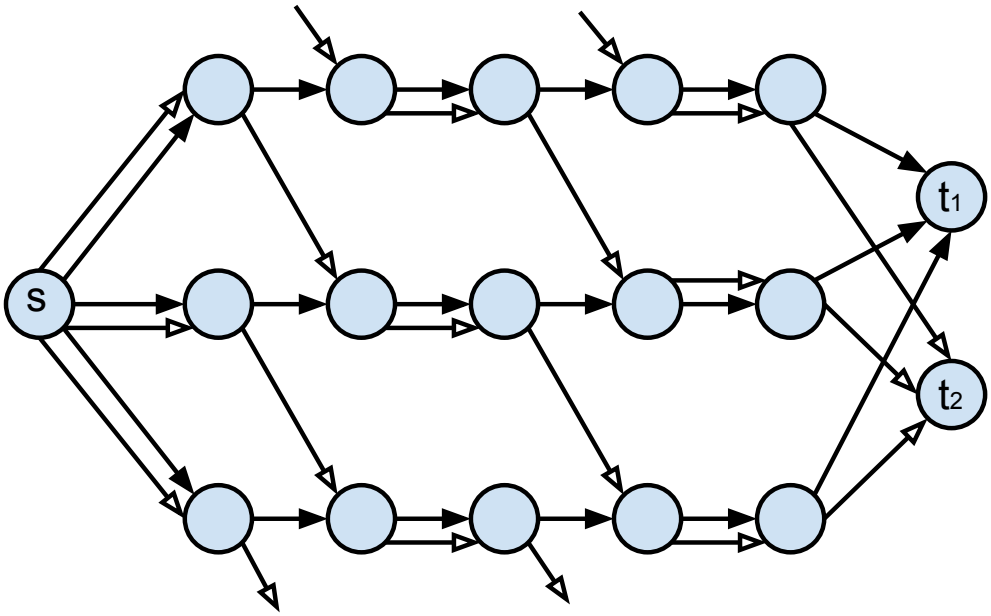
We will be showing that no algorithm with allocated input can prove the conjecture, even in the unweighted case.

Theorem 2. *There does not exist an algorithm with allocated input that takes a flow with allocations and produces an unsplittable flow that both respects these allocations and satisfies the conditions of Theorem 1.*

We know that Michel Goemans has been aware of this fact but to the best of our knowledge it is unpublished. The proof presented here is our own.

The main idea of the proof is that there may not exist an unsplittable flow that respects a given flow with allocations. Before we prove this we will discuss the implications of this result. Note that the fork matching idea from the previous section is an algorithm with allocated input. This is because we applied the binning technique to the edge nodes, and in order to create bins you need a fractional solution to the fork matching problem. In other words, we are starting with a fractional assignment of terminals to paths. Furthermore we are expecting our final solution to assign every terminal to a path that it was assigned to (with nonzero weight) in the fractional assignment. But this implies that we are starting with a flow with allocations and asking for an unsplittable flow that respects these allocations. By Theorem 2, this is impossible. More generally, we can also rule out a number of algorithms that attempt to extend the binning technique to general graphs. A natural first step for a binning algorithm would be to find an arbitrary flow with allocations and to then bin each edge based on these fractional allocations. We would then try to pick one demand from each bin to route across the edge. This runs into the same problem as above because this would yield a flow that respects the input allocations. This is good evidence that we will need more than the ideas from the 2-tier proof in order to prove the general case of the conjecture.

Now we prove Theorem 2 with the following counter-example, which consists of a graph and a flow with allocations for which there does not exist an unsplittable flow that both respects these allocations and satisfies the conditions of Theorem 1.



For clarity some edges go off the bottom of the figure and wrap around to the top. The terminals t_1 and t_2 each have unit-demand. The black arrows show 3 disjoint paths from s to t_1. Let the fractional flow with allocations assign t_1 to each of these paths with weight $\frac{1}{3}$. The white arrows show 3 disjoint paths from s to t_2. Let the fractional flow with allocations assign t_2 to each of these paths with weight $\frac{1}{3}$. Note that the flow on any edge is at most $\frac{2}{3}$ in the fractional solution. Theorem 1 demands that the flow on an edge can increase by at most $d_{max} = 1$, so our unsplittable flow is never allowed to route both commodities across the same edge. In order to make the unsplittable flow respect the allocations, we need to choose 1 of the 3 black paths for t_1 and 1 of the 3 white paths for t_2. However, no matter which paths we choose they will always share an edge, causing both commodities to be routed across the same edge. Therefore there is no unsplittable flow that both respects the allocations and satisfies the conditions of Theorem 1. This completes the proof of Theorem 2.

5 Conclusion

We have introduced the concept of unsplittable flows and described an open conjecture of Goemans. We have explained the proof of the conjecture for two special cases. The first was the unweighted case where all edges have cost zero, and the second was the weighted case but restricted to 2-tier graphs that capture a particular scheduling problem. We then discussed some of our own ideas for proving the conjecture in general. The key ideas of matching and binning from the 2-tier proof do not seem to extend to the general case. We justified this with some impossibility results. Namely, we showed that the fork matching problem does not necessarily have integral extreme points, and so the 2-tier proof does not generalize directly. More generally, we showed that any technique that starts with a fractional flow with allocations is doomed to fail for constructing a proof of the fully general case of the conjecture. This rules out the possibility of the binning technique from the 2-tier proof unless substantial modifications are made to it. An additional idea that we did not have time to explore here is the possibility of extending the algorithm for the unweighted case to the weighted case. This would perhaps involve finding particular alternating cycles that do not increase the overall cost. Investigation of this is left to future work.

References

[1] Dinitz, Yefim, Naveen Garg, and Michel X. Goemans. "On the single-source unsplittable flow problem." Combinatorica 19.1 (1999): 17-41.

[2] Shmoys, David B., and Éva Tardos. "An approximation algorithm for the generalized assignment problem." Mathematical Programming 62.1-3 (1993): 461-474.

[3] Martens, Maren, Fernanda Salazar, and Martin Skutella. "Convex combinations of single source unsplittable flows." Algorithms-ESA 2007. Springer Berlin Heidelberg, 2007. 395-406.