Constant Factor Approximation Algorithm for Weighted Flow Time on a Single Machine in Pseudo-polynomial time

Jatin Batra Naveen Garg Amit Kumar

Department of Computer Science and Engineering
IIT Delhi

***Abstract*—In the weighted flow-time problem on a single machine, we are given a set of n jobs, where each job has a processing requirement p_j, release date r_j and weight w_j. The goal is to find a preemptive schedule which minimizes the sum of weighted flow-time of jobs, where the flow-time of a job is the difference between its completion time and its released date. We give the first pseudo-polynomial time constant approximation algorithm for this problem. The algorithm also extends directly to the problem of minimizing the ℓ_p norm of weighted flow-times. The running time of our algorithm is polynomial in n, the number of jobs, and P, which is the ratio of the largest to the smallest processing requirement of a job. Our algorithm relies on a novel reduction of this problem to a generalization of the multi-cut problem on trees, which we call `Demand MultiCut` problem. Even though we do not give a constant factor approximation algorithm for the `Demand MultiCut` problem on trees, we show that the specific instances of `Demand MultiCut` obtained by reduction from weighted flow-time problem instances have more structure in them, and we are able to employ techniques based on dynamic programming. Our dynamic programming algorithm relies on showing that there are near optimal solutions which have nice smoothness properties, and we exploit these properties to reduce the size of DP table.**

I. Introduction

Scheduling jobs to minimize the average waiting time is one of the most fundamental problems in scheduling theory with numerous applications. We consider the setting where jobs arrive over time (i.e., have release dates), and need to be processed such that the average flow-time is minimized. The flow-time, F_j of a job j, is defined as the difference between its completion time, C_j, and release date, r_j. It is well known that for the case of single machine, the SRPT policy (Shortest Remaining Processing Time) gives an optimal algorithm for this objective.

In the weighted version of this problem, jobs have weights and we would like to minimize the weighted sum of flow-time of jobs. However, the problem of minimizing *weighted* flow-time (`WtdFlowTime`) turns out to be NP-hard and it has been widely conjectured that there should a constant factor approximation algorithm (or even PTAS) for it. In this paper, we make substantial progress towards this problem by giving the first constant factor approximation algorithm for this problem in pseudo-polynomial time. More formally, we prove the following result.

Theorem I.1. *There is a constant factor approximation algorithm for* `WtdFlowTime` *where the running time of the algorithm is polynomial in n and P. Here, n denotes the number of jobs in the instance, and P denotes the ratio of the largest to the smallest processing time of a job in the instance respectively.*

We obtain this result by reducing `WtdFlowTime` to a generalization of the multi-cut problem on trees, which we call `Demand MultiCut`. The `Demand MultiCut` problem is a natural generalization of the multi-cut problem where edges have sizes and costs, and input paths (between terminal pairs) have demands. We would like to select a minimum cost subset of edges such that for every path in the input, the total size of the selected edges in the path is at least the demand of the path. When all demands and sizes are 1, this is the usual multi-cut problem. The natural integer program for this problem has the property that all non-zero entries in any column of the constraint matrix are the same. Such integer programs, called *column restricted covering integer programs*, were studied by Chakrabarty et al. [7]. They showed that one can get a constant factor approximation algorithm for `Demand MultiCut` provided one could prove that the integrality gap of the natural LP relaxations for the following two special cases is constant – (i) the version where the constraint matrix has 0-1 entries only, and (ii) the priority version, where paths and edges in the tree have priorities (instead of sizes and demands respectively), and we want to pick minimum cost subset of edges such that for each path, we pick at least one edge in it of priority which is at least the priority of this path. Although the first problem turns out to be easy, we do not know how to

2575-8454/18/$31.00 ©2018 IEEE
DOI 10.1109/FOCS.2018.00079

round the LP relaxation of the priority version. This is similar to the situation faced by Bansal and Pruhs [4], where they need to round the priority version of a geometric set cover problem. They appeal to the notion of shallow cell complexity [8] to get an $O(\log\log P)$-approximation for this problem. It turns out the shallow cell complexity of the priority version of `Demand MultiCut` is also unbounded (depends on the number of distinct priorities) [8], and so it is unlikely that this approach will yield a constant factor approximation.

However, the specific instances of `Demand MultiCut` produced by our reduction have more structure, namely each node has at most 2 children, each path goes from an ancestor to a descendant, and the tree has $O(\log(nP))$ depth if we shortcut all degree 2 vertices. We show that one can effectively use dynamic programming techniques for such instances. We show that there is a near optimal solution which has nice "smoothness" properties so that the dynamic programming table can manage with storing small amount of information.

A. Related Work

There has been a lot of work on the `WtdFlowTime` problem on a single machine, though polynomial time constant factor approximation algorithm has remained elusive. Bansal and Dhamdhere [1] gave an $O(\log W)$-competitive on-line algorithm for this problem, where W is the ratio of the maximum to the minimum weight of a job. They also gave a semi-online (where the algorithm needs to know the parameters P and W in advance) $O(\log(nP))$-competitive algorithm for `WtdFlowTime`, where P is the ratio of the largest to the smallest processing time of a job. Chekuri et al. [10] gave a semi-online $O(\log^2 P)$-competitive algorithm.

Recently, Bansal and Pruhs [4] made significant progress towards this problem by giving an $O(\log\log P)$-approximation algorithm. In fact, their result applies to a more general setting where the objective function is $\sum_j f_j(C_j)$, where $f_j(C_j)$ is any monotone function of the completion time C_j of job j. Their work, along with a constant factor approximation for the generalized caching problem [5], implies a constant factor approximation algorithm for this setting when all release dates are 0. Chekuri and Khanna [9] gave a quasi-PTAS for this problem, where the running time was $O(n^{O_\epsilon(\log W \log P)})$. In the special case of stretch metric, where $w_j = 1/p_j$, PTAS is known [6], [9]. The problem of minimizing (unweighted) ℓ_p norm of flow-times was studied by Im and Moseley [12] who gave a constant factor approximation in polynomial time.

In the speed augmentation model introduced by Kalyanasundaram and Pruhs [13], the algorithm is given $(1+\varepsilon)$-times extra speed than the optimal algorithm. Bansal and Pruhs [3] showed that Highest Density First (HDF) is $O(1)$-competitive for weighted ℓ_p norms of flow-time for all values of $p \geq 1$.

The multi-cut problem on trees is known to be NP-hard, and a 2-approximation algorithm was given by Garg et al. [11]. As mentioned earlier, Chakrabarty et al. [7] gave a systematic study of column restricted covering integer programs (see also [2] for follow-up results). The notion of shallow cell complexity for 0-1 covering integer programs was formalized by Chan et al. [8], where they relied on and generalized the techniques of Varadarajan [14].

II. Preliminaries

An instance of the `WtdFlowTime` problem is specified by a set of n jobs. Each job has a processing requirement p_j, weight w_j and release date r_j. We assume wlog that all of these quantities are integers, and let P denote the ratio of the largest to the smallest processing requirement of a job. We divide the time line into unit length *slots* – we shall often refer to the time slot $[t, t+1]$ as slot t. A feasible schedule needs to process a job j for p_j units after its release date. Note that we allow a job to be preempted. The weighted flow-time of a job is defined as $w_j \cdot (C_j - r_j)$, where C_j is the slot in which the job j finishes processing. The objective is to find a schedule which minimizes the sum over all jobs of their weighted flow-time.

Note that any schedule would occupy exactly $T = \sum_j p_j$ slots. We say that a schedule is *busy* if it does not leave any slot vacant even though there are jobs waiting to be finished. We can assume that the optimal schedule is a busy schedule (otherwise, we can always shift some processing back and improve the objective function). We also assume that any busy schedule fills the slots in $[0, T]$ (otherwise, we can break it into independent instances satisfying this property).

We shall also consider a generalization of the multi-cut problem on trees, which we call the `Demand MultiCut` problem. Here, edges have cost and size, and demands are specified by ancestor-descendant paths. Each such path has a demand, and the goal is to select a minimum cost subset of edges such that for each path, the total size of selected edges in the path is at least the demand of this path.

In Section II-A, we describe a well-known integer program for `WtdFlowTime`. This IP has variables $x_{j,t}$ for every job j, and time $t \geq r_j$, and it is supposed to be 1 if j completes processing after time t. The constraints in the IP consist of several covering constraints. However, there is an additional complicating factor that

$x_{j,t} \le x_{j,t-1}$ must hold for all $t \ge r_j$. To get around this problem, we propose a different IP in Section III. In this IP, we define variables of the form $y(j,S)$, where S are exponentially increasing intervals starting from the release date of j. This variable indicates whether j is alive during the entire duration of S. The idea is that if the flow-time of j lies between 2^i and 2^{i+1}, we can count 2^{i+1} for it, and say that j is alive during the entire period $[r_j + 2^i, r_j + 2^{i+1}]$. Conversely, if the variable $y(j,S)$ is 1 for an interval of the form $[r_j + 2^i, r_j + 2^{i+1}]$, we can assume (at a factor 2 loss) that it is also alive during $[r_j, r_j + 2^i]$. This allows us to decouple the $y(j,S)$ variables for different S. By an additional trick, we can ensure that these intervals are laminar for different jobs. From here, the reduction to the `Demand MultiCut` problem is immediate (see Section IV for details). In Section V, we show that the specific instances of `Demand MultiCut` obtained by such reductions have additional properties. We use the property that the tree obtained from shortcutting all degree two vertices is binary and has $O(\log(nP))$ depth. We shall use the term *segment* to define a maximal degree 2 (ancestor-descendant) path in the tree. So the property can be re-stated as – any root to leaf path has at most $O(\log(nP))$ segments. We give a dynamic programming algorithm for such instances. In the DP table for a vertex in the tree, we will look at a sub-instance defined by the sub-tree below this vertex. However, we also need to maintain the "state" of edges above it, where the state means the ancestor edges selected by the algorithm. This would require too much book-keeping. We use two ideas to reduce the size of this state – (i) We first show that the optimum can be assumed to have certain smoothness properties, which cuts down on the number of possible configurations. The smoothness property essentially says that the cost spent by the optimum on a segment does not vary by more than a constant factor as we go to neighbouring segments, (ii) If we could spend twice the amount spent by the algorithm on a segment S, and select low density edges, we could ignore the edges in a segment S' lying above S in the tree.

A. An integer program

We describe an integer program for the `WtdFlowTime` problem. This is well known (see e.g. [4]), but we give details for sake of completeness. We will have binary variables $x_{j,t}$ for every job j and time t, where $r_j \le t \le T$. This variable is meant to be 1 iff j is *alive* at time t, i.e., its completion time is at least t. Clearly, the objective function is $\sum_j \sum_{t \in [r_j,T]} w_j x_{j,t}$. We now specify the constraints of the integer program. Consider a time interval $I = [s,t]$, where $0 \le s \le t \le T$, and s and t are integers. Let $l(I)$ denote the length of this time interval, i.e., $t - s$. Let $J(I)$ denote the set of jobs released during I, i.e., $\{j : r_j \in I\}$, and $p(J(I))$ denote the total processing time of jobs in $J(I)$. Clearly, the total volume occupied by jobs in $J(I)$ beyond I must be at least $p(J(I)) - l(I)$. Thus, we get the following integer program: (IP1)

$$\min \sum_j \sum_{t \in [r_j,T]} w_j x_{j,t} \tag{1}$$

$$\sum_{j \in J(I)} x_{j,t} p_j \ge p(J(I)) - l(I)$$

$$\text{for all } I = [s,t], 0 \le s \le t \le T \tag{2}$$

$$x_{j,t} \le x_{j,t-1} \quad \text{for all } j,t,\ r_j < t \le T \tag{3}$$

$$x_{j,t} \in \{0,1\} \text{ for all } j,t$$

It is easy to see that this is a relaxation – given any schedule, the corresponding $x_{j,t}$ variables will satisfy the constraints mentioned above, and the objective function captures the total weighted flow-time of this schedule. The converse is also true – given any solution to the above integer program, there is a corresponding schedule of the same cost.

Theorem II.1. *Suppose $x_{j,t}$ is a feasible solution to (IP1). Then, there is a schedule for which the total weighted flow-time is equal to the cost of the solution $x_{j,t}$.*

III. A Different Integer Program

We now write a weaker integer program, but it has more structure in it. We first assume that T is a power of 2 – if not, we can pad the instance with a job of zero weight (this will increase the ratio P by at most a factor n only). Let T be 2^ℓ. We now divide the time line into nested dyadic segments. A dyadic segment is an interval of the form $[i \cdot 2^s, (i+1) \cdot 2^s]$ for some non-negative integers i and s (we shall use segments to denote such intervals to avoid any confusion with intervals used in the integer program). For $s = 0, \ldots, \ell$, we define $\mathcal{S}_s$ as the set of dyadic segments of length 2^s starting from 0, i.e., $\{[0,2^s], [2^s, 2 \cdot 2^s], \ldots, [i \cdot 2^s, (i+1) \cdot 2^s], \ldots, [T - 2^s, T]\}$. Clearly, any segment of $\mathcal{S}_s$ is contained inside a unique segment of $\mathcal{S}_{s+1}$. Now, for every job j we shall define a sequence of dyadic segments $\texttt{Seg}(j)$. The sequence of segments in $\texttt{Seg}(j)$ partition the interval $[r_j, T]$. The construction of $\texttt{Seg}(j)$ is described in Figure 1 (also see the example in Figure 2). It is easy to show by induction on s that the parameter t at the beginning of iteration s in Step 2 of the algorithm is a multiple of 2^s. Therefore, the segments added during the iteration for s belong to $\mathcal{S}_s$. Although we do not specify for how long we run the for loop in Step 2, we stop when t reaches T (this will always happen because t takes values from the set of end-points in the segments in $\cup_s \mathcal{S}_s$). Therefore the set of segments in $\texttt{Seg}(j)$ are disjoint and cover $[r_j,T]$.

For a job j and segment $S \in \texttt{Seg}(j)$, we shall refer to the tuple (j,S) as a *job-segment*. For a time t, we say that $t \in (j,S)$ (or (j,S) contains t) if $[t,t+1] \subseteq S$. We now show a crucial nesting property of these segments.

Lemma III.1. *Suppose (j, S) and (j', S') are two job-segments such that there is a time t for which $t \in (j, S)$ and $t \in (j', S')$. Suppose $r_j \leq r_{j'}$, and $S \in \mathcal{S}_s, S \in \mathcal{S}_{s'}$. Then $s \geq s'$.*

We now write a new IP. The idea is that if a job j is alive at some time t, then we will keep it alive during the entire duration of the segment in $\texttt{Seg}(j)$ containing t. Since the segments in $\texttt{Seg}(j)$ have lengths in exponentially increasing order (except for two consecutive segments), this will not increase the weighted flow-time by more than a constant factor. For each job segment (j,S) we have a binary variable $y(j,S)$, which is meant to be 1 iff the job j is alive during the entire duration S. For each job segment (j,S), define its weight $w(j,S)$ as $w_j \cdot l(S)$ – this is the contribution towards weighted flow-time of j if j remains alive during the entire segment S. We get the following integer program (IP2):

$$\min \sum_j \sum_s w(j,S) y(j,S) \tag{4}$$

$$\sum_{(j,S): j \in J(I), t \in (j,S)} y(j,S) p_j \geq p(J(I)) - l(I)$$

$$\text{for all } I = [s,t], 0 \leq s \leq t \leq T \tag{5}$$

$$y(j,S) \in \{0,1\} \quad \text{for all } (j,S)$$

Observe that for any interval I, the constraint (5) for I has precisely one job segment for every job which gets released in I. Another interesting feature of this IP is that we do not have constraints corresponding to (3), and so it is possible that $y(j,S) = 1$ and $y(j,S') = 0$ for two job segments (j,S) and (j,S') even though S' appears before S in $\texttt{Seg}(j)$. We now relate the two integer programs.

Lemma III.2. *Given a solution x for (IP1), we can construct a solution for (IP2) of cost at most 8 times the cost of x. Similarly, given a solution y for (IP2), we can construct a solution for (IP1) of cost at most 4 times the cost of y.*

The above lemma states that it is sufficient to find a solution for (IP2). Note that (IP2) is a covering problem. It is also worth noting that the constraints (5) need to be written only for those intervals $[s, t]$ for which a job segment starts or ends at s or t. Since the number of job segments is $O(n \log T) = O(n \log(nP))$, it follows that (IP2) can be turned into a polynomial size integer program.

IV. Reduction to `Demand MultiCut` on Trees

We now show that (IP2) can be viewed as a covering problem on trees. We define the covering problem, which we call Demand Multi-cut(`Demand MultiCut`) on trees. An instance $\mathcal{I}$ of this problem consists of a tuple $(\mathcal{T},\mathcal{P},c,p,d)$, where $\mathcal{T}$ is a rooted tree, and $\mathcal{P}$ consists of a set of ancestor-descendant paths. Each edge e in $\mathcal{T}$ has a cost c_e and size p_e. Each path $P in \mathcal{P}$ has a demand $d(P)$. Our goal is to pick a minimum cost subset of vertices V' such that for every path $P \in \mathcal{P}$, the set of vertices in $V' \cap P$ have total size at least $d(P)$.

We now reduce `WtdFlowTime` to `Demand MultiCut` on trees. Consider an instance $\mathcal{I}'$ of `WtdFlowTime` consisting of a set of jobs J. We reduce it to an instance $\mathcal{I} = (\mathcal{T},\mathcal{P},c,p,d)$ of `Demand MultiCut`. In our reduction, $\mathcal{T}$ will be a forest instead of a tree, but we can then consider each tree as an independent problem instance of `Demand MultiCut`.

We order the jobs in J according to release dates (breaking ties arbitrarily) – let $\prec_J$ be this total ordering (so, $j \prec_J j'$ implies that $r_j \leq r_{j'}$). We now define the forest $\mathcal{T}$. The vertex set of $\mathcal{T}$ will consist of all job segments (j,S). For such a vertex (j,S), let j' be the job immediately preceding j in the total order $\prec_J$. Since the job segments in $\texttt{Seg}(j')$ partition $[r_{j'}, T]$, and $r_{j'} \leq r_j$, there is a pair (j',S') in $\texttt{Seg}(j')$ such that S' intersects S, and so contains S, by Lemma III.1. We define (j',S') as the parent of (j,S). It is easy to see that this defines a forest structure, where the root vertices correspond to (j, S), with j being the first job in $\prec$. Indeed, if $(j_1,S_1),(j_2,S_2),\ldots,(j_k,S_k)$ is a sequence of nodes with (j_i, S_i) being the parent of (j_{i+1},S_{i+1}), then $j_1 \prec_J j_2 \prec_J \cdots \prec_J j_k$, and so no node in this sequence can be repeated.

For each tree in this forest $\mathcal{T}$ with the root vertex being (j,S), we add a new root vertex r and make it the parent of (j,S). We now define the cost and size of each edge. Let $e = (v_1,v_2)$ be an edge in the tree, where v_1 is the parent of v_2. Let v_2 correspond to the job segment (j,S). Then $p_e = p_j$ and $c_e = w_e \cdot l(S)$. In other words, picking edge e corresponds to selecting the job segment (j,S).

Now we define the set of paths $\mathcal{P}$. For each constraint (5) in (IP2), we will add one path in $\mathcal{P}$. We first observe the following property. Fix an interval $I = [s,t]$ and consider the constraint (5) corresponding

```
Algorithm FormSegments($j$)
1. Initialize $t \leftarrow r_j$.
2. For $s=0,1,2,...,$
      (i) If $t$ is a multiple of $2^{s+1}$,
            add the segments (from the set $\mathcal{S}_s$) $[t,t+2^s],[t+2^s,t+2^{s+1}]$ to Seg($j$)
            update $t \leftarrow t+2^{s+1}$.
      (ii) Else add the segment (from the set $\mathcal{S}_s$) $[t,,t+2^s]$ to Seg($j$).
            update $t \leftarrow t+2^s$.
```

Figure 1. Forming $\mathtt{Seg}(j)$.

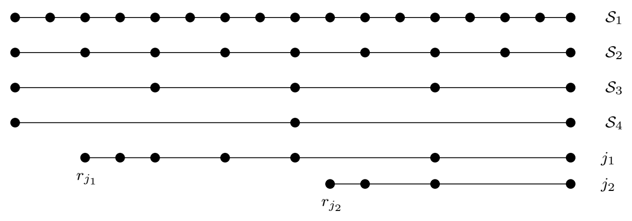

Figure 2. The dyadic segments $\mathcal{S}_1,...,\mathcal{S}_4$ and the corresponding $\mathtt{Seg}(j_1)$,$\mathtt{Seg}(j_2)$ for two jobs j_1,j_2

to it. Let V_I be the vertices in $\mathcal{T}$ corresponding to the job segments appearing in the LHS of this constraint.

Lemma IV.1. *The vertices in V_I form a path in $\mathcal{T}$ from an ancestor to a descendant.*

Let the vertices in V_I be $v_1,...,v_k$ arranged from ancestor to descendant. Let v_0 be the parent of v_1 (this is the reason why we added an extra root to each tree – just in case v_1 corresponds to the first job in $\prec_J$, it will still have a parent). We add a path $P_I = v_0,v_1,...,v_k$ to $\mathcal{P}$ – Lemma IV.1 guarantees that this will be an ancestor-descendant path. The demand $d(P)$ of this path is the quantity in the RHS of the corresponding constraint (5) for the interval I. The following claim is now easy to check.

Claim IV.2. *Given a solution E to the* `Demand MultiCut` *instance $\mathcal{I}$, there is a solution to (IP2) for the instance $\mathcal{I}'$ of the same objective function value as that of E.*

This completes the reduction from `WtdFlowTime` to `Demand MultiCut`. This reduction is polynomial time because number of vertices in $\mathcal{T}$ is equal to the number of job segments, which is $O(n\log(nP))$. Each path in $\mathcal{P}$ goes between any two vertices in $\mathcal{T}$, and there is no need to have two paths between the same pair of vertices. Therefore the size of the instance $\mathcal{I}$ is polynomial in the size of the instance $\mathcal{I}'$ of `WtdFlowTime`.

V. Approximation Algorithm for the `Demand MultiCut` problem

In this section we give a constant factor approximation algorithm for the special class of `Demand MultiCut` problems which arise in the reduction from `WtdFlowTime`. To understand the special structure of such instances, we begin with some definitions. Let $\mathcal{I} = (\mathcal{T}, \mathcal{P}, c, p, d)$ be an instance of `Demand MultiCut`. The *density* ρ_e of an edge e is defined as the ratio c_e/p_e. Let $\mathtt{red}(\mathcal{T})$ denote the tree obtained from $\mathcal{T}$ by short-cutting all non-root degree 2 vertices (see Figure 3 for an example). There is a clear correspondence between the vertices of $\mathtt{red}(\mathcal{T})$ and the non-root vertices in $\mathcal{T}$ which do not have degree 2. In fact, we shall use $V(\mathtt{red}(\mathcal{T}))$ to denote the latter set of vertices. The reduced height of $\mathcal{T}$ is defined as the height of $\mathtt{red}(\mathcal{T})$. In this section, we prove the following result. We say that a (rooted) tree is binary if every node has at most 2 children.

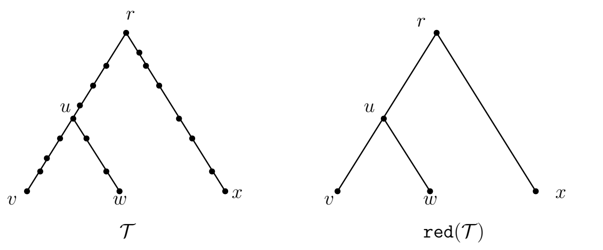

Figure 3. Tree $\mathcal{T}$ and the corresponding tree $\mathtt{red}(\mathcal{T})$. Note that the vertices in $\mathtt{red}(\mathcal{T})$ are also present in $\mathcal{T}$, and the segments in $\mathcal{T}$ correspond to edges in $\mathtt{red}(\mathcal{T})$. The tree $\mathcal{T}$ has 4 segments, e.g., the path between r and u.

Theorem V.1. *There is a constant factor approximation algorithm for instances $\mathcal{I} = (\mathcal{T},\mathcal{P},c,p,d)$ of* `Demand MultiCut` *where $\mathcal{T}$ is a binary tree. The running time of this algorithm is $poly(n,2^{O(H)},\rho_{\max}/\rho_{\min})$, where n denotes the number of nodes in $\mathcal{T}$, H denotes the reduced height of $\mathcal{T}$, and $\rho_{\max}$ and $\rho_{\min}$ are the maximum* *and the minimum density of an edge in $\mathcal{T}$ respectively.*

Remark: In the instance $\mathcal{I}$ above, some edges may have 0 size. These edges are not considered while defining $\rho_{\max}$ and ρ_{min}.

One can show that this theorem implies the main result in Theorem I.1.

We now prove Theorem V.1 in rest of the paper.

A. Some Special Cases

To motivate our algorithm, we consider some special cases first. Again, fix an instance $\mathcal{I} = (\mathcal{T},\mathcal{P},c,p,d)$ of `Demand MultiCut`. Recall that the tree $\texttt{red}(\mathcal{T})$ is obtained by short-cutting all degree 2 vertices in $\mathcal{T}$. Each edge in $\texttt{red}(\mathcal{T})$ corresponds to a path in $\mathcal{T}$ – in fact, there are maximal paths in $\mathcal{T}$ for which all internal nodes have degree 2. We call such paths *segments* (to avoid confusion with paths in $\mathcal{P}$). See Figure 3 for an example. Thus, there is a 1-1 correspondence between edges in $\texttt{red}(\mathcal{T})$ and segments in $\mathcal{T}$. Recall that every vertex in $\texttt{red}(\mathcal{T})$ corresponds to a vertex in $\mathcal{T}$ as well, and we will use the same notation for both the vertices.

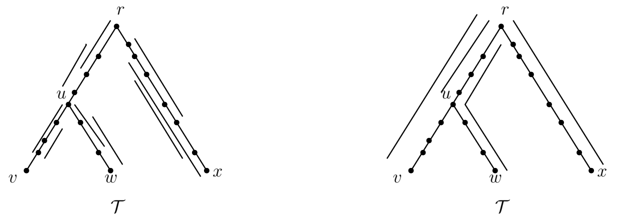

Figure 4. The left instance represents a segment confined instance whereas the right one is a segment spanning instance.

1) Segment Confined Instances: The instance $\mathcal{I}$ is said to be *segment confined* if all paths in $\mathcal{P}$ are confined to one segment, i.e., for every path $P \in \mathcal{P}$, there is a segment S in $\mathcal{T}$ such that the edges of P are contained in S.

In this section, we show that one can obtain constant factor polynomial time approximation algorithms for such instances. In fact, this result follows from prior work on column restricted covering integer programs [7].

Since each path in $\mathcal{P}$ is confined to one segment, we can think of this instance as several independent instances, one for each segment. For a segment S, let $\mathcal{I}_S$ be the instance obtained from $\mathcal{I}$ by considering edges in S only and the subset $\mathcal{P}_S \subseteq \mathcal{P}$ of paths which are contained in S. We show how to obtain a constant factor approximation algorithm for $\mathcal{I}_S$ for a fixed segment S.

Let the edges in S (in top to down order) be $e_1,\ldots,e_m$. The following integer program (IP3) captures the `Demand MultiCut` problem for $\mathcal{I}_S$:

$$\min \sum_{e \in S} c_e x_e \tag{6}$$

$$\sum_{e \in P} p_e x_e \geq d(P) \qquad \text{for all paths } P \in \mathcal{P}_S \tag{7}$$

$$x_e \in \{0,1\} \quad \text{for all } e \in S \tag{8}$$

Note that this is a covering integer program (IP) where the coefficient of x_e in each constraint is either 0 or p_e. Such an IP comes under the class of Column Restricted Covering IP as described in [7]. Chakrabarty et al. [7] show that one can obtain a constant factor approximation algorithm for this problem provided one can prove that the integrality gaps of the corresponding LP relaxations for the following two special class of problems are constant: (i) 0-1 instances, where the p_e values are either 0 or 1, (ii) priority versions, where paths in $\mathcal{P}$ and edges have priorities (which can be thought of as positive integers), and the selected edges satisfy the property that for each path $P \in \mathcal{P}_S$, we selected at least one edge in it of priority at least that of P (it is easy to check that this is a special case of `Demand MultiCut` problem by assigning exponentially increasing demands to paths of increasing priority, and similarly for edges).

Consider the class of 0-1 instances first. We need to consider only those edges for which p_e is 1 (contract the edges for which p_e is 0). Now observe that the constraint matrix on the LHS in (IP3) has consecutive ones property (order the paths in $\mathcal{P}_S$ in increasing order of left end-point and write the constraints in this order). Therefore, the LP relaxation has integrality gap of 1.

Rounding the Priority Version We now consider the priority version of this problem. For each edge $e \in S$, we now have an associated priority p_e (instead of size), and each path in $\mathcal{P}$ also has a priority demand $p(P)$, instead of its demand. We need to argue about the integrality gap of the following LP relaxation:

$$\min \sum_{e \in S} c_e x_e \tag{9}$$

$$\sum_{e \in P : p_e \geq p(P)} x_e \geq 1 \qquad \text{for all paths } P \in \mathcal{P}_S \tag{10}$$

$$x_e \geq 0 \quad \text{for all } e \in S \tag{11}$$

We shall use the notion of *shallow cell complexity* used in [8]. Let A be the constraint matrix on the LHS above. We first notice the following property of A.

Claim V.2. *Let $A^\star$ be a subset of s columns of A. For a parameter $k, 0 \leq k \leq s$, there are at most $k^2 s$ distinct rows in $A^\star$ with k or fewer 1's (two rows of $A^\star$ are distinct iff they are not same as row vectors).*

In the notation of [8], the shallow cell complexity of this LP relaxation is $f(s,k)=sk^2$. It now follows from Theorem 1.1 in [8] that the integrality gap of the LP relaxation for the priority version is a constant. Thus we obtain a constant factor approximation algorithm for segment restricted instances.

2) Segment Spanning Instances on Binary Trees: We now consider instances $\mathcal{I}$ for which each path $P \in \mathcal{P}$ starts and ends at the end-points of a segment, i.e., the starting or ending vertex of P belongs to the set of vertices in $\mathtt{red}(\mathcal{T})$. An example is shown in Figure 4. Although we will not use this result in the algorithm for the general case, many of the ideas will get extended to the general case. We will use dynamic programming. For a vertex $v \in \mathtt{red}(\mathcal{T})$, let $\mathcal{T}_v$ be the sub-tree of $\mathcal{T}$ rooted below v (and including v). Let $\mathcal{P}_v$ denote the subset of $\mathcal{P}$ consisting of those paths which contain at least one edge in $\mathcal{T}_v$. By scaling the costs of edges, we will assume that the cost of the optimal solution lies in the range $[1,n]$ – if $c_{\max}$ is the maximum cost of an edge selected by the optimal algorithm, then its cost lies in the range $[c_{\max}, nc_{\max}]$.

Before stating the dynamic programming algorithm, we give some intuition for the DP table. We will consider sub-problems which correspond to covering paths in $\mathcal{P}_v$ by edges in $\mathcal{T}_v$ for every vertex $v \in \mathtt{red}(\mathcal{T})$. However, to solve this sub-problem, we will also need to store the edges in $\mathcal{T}$ which are ancestors of v and are selected by our algorithm. Storing all such subsets would lead to too many DP table entries. Instead, we will work with the following idea – for each segment S, let $B^{\mathtt{opt}}(S)$ be the total cost of edges in S which get selected by an optimal algorithm. If we know $B^{\mathtt{opt}}(S)$, then we can decide which edges in S can be picked. Indeed, the optimal algorithm will solve a knapsack cover problem – for the segment S, it will pick edges of maximum total size subject to the constraint that their total cost is at most $B^{\mathtt{opt}}(S)$ (note that we are using the fact that every path in $\mathcal{P}$ which includes an edge in S must include all the edges in S). Although knapsack cover is NP-hard, here is a simple greedy algorithm which exceeds the budget $B^{\mathtt{opt}}(S)$ by a factor of 2, and does as well as the optimal solution (in terms of total size of selected edges) – order the edges in S whose cost is at most $B^{\mathtt{opt}}(S)$ in order of increasing density. Keep selecting them in this order till we exceed the budget $B^{\mathtt{opt}}(S)$. Note that we pay at most twice of $B^{\mathtt{opt}}(S)$ because the last edge will have cost at most $B^{\mathtt{opt}}(S)$. The fact that the total size of selected edges is at least that of the corresponding optimal value follows from standard greedy arguments.

Therefore, if $S_1, \ldots, S_k$ denote the segments which lie above v (in the order from the root to v), it will suffice if we store $B^{\mathtt{opt}}(S_1), \ldots, B^{\mathtt{opt}}(S_k)$ with the DP table entry for v. We can further cut-down the search space by assuming that each of the quantities $B^{\mathtt{opt}}(S)$ is a power of 2 (we will lose only a multiplicative 2 in the cost of the solution). Thus, the total number of possibilities for $B^{\mathtt{opt}}(S_1), \ldots, B^{\mathtt{opt}}(S_k)$ is $O(\log^k n)$, because each of the quantities $B^{\mathtt{opt}}(S_i)$ lies in the range $[1, 2n]$ (recall that we had assumed that the optimal value lies in the range $[1, n]$ and now we are rounding this to power of 2). This is at most $2^{O(H \log\log n)}$, which is still not polynomial in n and $2^{O(H)}$. We can further reduce this by assuming that for any two consecutive segments S_i, S_{i+1}, the quantities $B^{\mathtt{opt}}(S_i)$ and $B^{\mathtt{opt}}(S_{i+1})$ differ by a factor of at most 8 – it is not clear why we can make this assumption, but we will show later that this does leads to a constant factor loss only. We now state the algorithm formally.

Dynamic Programming Algorithm

We first describe the greedy algorithm outlined above. The algorithm `GreedySelect` is given in Figure 5.

For a vertex $v \in \mathtt{red}(\mathcal{T})$, define the *reduced depth* of v as its at depth in $\mathtt{red}(\mathcal{T})$ (root has reduced depth 0). We say that a sequence $B_1, \ldots, B_k$ is a *valid state sequence* at a vertex v in $\mathtt{red}(\mathcal{T})$ with reduced depth k if it satisfies the following conditions:

- For all $i=1,\ldots,k$, B_i is a power of 2 and lies in the range $[1,2n]$.
- For any $i=1,\ldots,k-1$, B_i/B_{i+1} lies in the range $[1/8,8]$.

If $S_1,\ldots,S_k$ is the sequence of segments visited while going from the root to v, then B_i will correspond to S_i.

Consider a vertex $v \in \mathtt{red}(\mathcal{T})$ at reduced depth k, and a child w of v in $\mathtt{red}(\mathcal{T})$ (at reduced depth $k+1$). Let $\Lambda_v = (B_1,\ldots,B_k)$ and $\Lambda_w = (B'_1,\ldots,B'_{k+1})$ be valid state sequences at these two vertices respectively. We say that Λ_w is an *extension* of Λ_v if $B_i = B'_i$ for $i=1,\ldots,k$. In the dynamic program, we maintain a table entry $T[v,\Gamma_v]$ for each vertex v in $\mathtt{red}(\mathcal{T})$ and valid state sequence Γ_v at v. Informally, this table entry stores the following quantity. Let $S_1,\ldots,S_k$ be the segments from the root to the vertex v. This table entry stores the minimum cost of a subset E' of edges in $\mathcal{T}_v$ such that $E' \cup G(v)$ is a feasible solution for the paths in $\mathcal{P}_v$, where $G(v)$ is the union of the set of edges selected by `GreedySelect` in the segments $S_1,\ldots,S_k$ with budgets $B_1,\ldots,B_k$ respectively.

The algorithm is described in Figure 6. We first compute the set $G(v)$ as outlined above. Let the children of v in the tree $\mathtt{red}(\mathcal{T})$ be w_1 and w_2. Let the segments corresponding to (v,w_1) and (v,w_2) be

```
Algorithm GreedySelect:
  Input: A segment $S$ in $\mathcal{T}$ and a budget $B$.
  1. Initialize a set $G$ to emptyset.
  2. Arrange the edges in $S$ of cost at most $B$ in ascending order of density.
  3. Keep adding these edges to $G$ till their total cost exceeds $B$.
  4. Output $G$.
```

Figure 5. Algorithm `GreedySelect` for selecting edges in a segment S with a budget B.

S^1_{k+1} and S^2_{k+1} respectively. For both these children, we find the best extension of Γ_v. For the node w_r, we try out all possibilities for the budget B^r_{k+1} for the segment S^r_{k+1}. For each of these choices, we select a set of edges in S^r_{k+1} as given by `GreedySelect` and lookup the table entry for w_r and the corresponding state sequence. We pick the choice for B^r_{k+1} for which the combined cost is smallest (see line 7(i)(c)).

We would like to remark that for any $v \in \texttt{red}(\mathcal{T})$, the number of possibilities for a valid state sequence is bounded by $2^{O(H)} \cdot \log n$. Indeed, there are $O(\log n)$ choices for B_1, and given B_i, there are only 7 choices for B_{i+1} (since B_{i+1}/B_i is a power of 2 and lies in the range $[1/8, 8]$). Therefore, the algorithm has running time polynomial in n and $2^{O(H)}$.

B. General Instances on Binary Trees

We now consider general instances of `Demand MultiCut` on binary trees. We can assume that every path $P \in \mathcal{P}$ contains at least one vertex of $\texttt{red}(\mathcal{T})$ as an internal vertex. Indeed, we can separately consider the instance consisting of paths in $\mathcal{P}$ which are contained in one of the segments – this will be a segment confined instance as in Section V-A1. We can get a constant factor approximation for such an instance.

We will proceed as in the previous section, but now it is not sufficient to know the total cost spent by an optimal solution in each segment. For example, consider a segment S which contains two edges e_1 and e_2; and e_1 has low density, whereas e_2 has high density. Now, we would prefer to pick e_1, but it is possible that there are paths in $\mathcal{P}$ which contain e_2 but do not contain e_1. Therefore, we cannot easily determine whether we should prefer picking e_1 over picking e_2. However, if all edges in S had the same density, then this would not be an issue. Indeed, given a budget B for S, we would proceed as follows – starting from each of the end-points, we will keep selecting edges of cost at most B till their total cost exceeds B. The reason is that all edges are equivalent in terms of cost per unit size, and since each path in $\mathcal{P}$ contains at least one of the end-points of S, we might as well pick edges which are closer to the end-points. Of course, edges in S may have varying density, and so, we will now need to know the budget spent by the optimum solution for each of the possible density values. We now describe this notion more formally.

Algorithm Description We first assume that the density of any edge is a power of 128 – we can do this by scaling the costs of edges by factors of at most 128. We say that an edge e is of *density class* τ if it's density is 128^τ. Let $\tau_{\max}$ and $\tau_{\min}$ denote the maximum and the minimum density class of an edge respectively. Earlier, we had specified a budget $B(S)$ for each segment S above v while specifying the state at v. Now, we will need to store more information at every such segment. We shall use the term *cell* to refer to a pair (S,τ), where S is a segment and τ is a density class[1]. Given a cell (S,τ) and a budget B, the algorithm `GreedySelect` in Figure 7 describes the algorithm for selecting edges of density class τ from S. As mentioned above, this procedure ensures that we pick enough edges from both end-points of S. The only subtlety is than in Step 4, we allow the cost to cross $2B$ – the factor 2 is for technical reasons which will become clear later. Note that in Step 4 (and similarly in Step 5) we could end up selecting edges of total cost up tp $3B$ because each selected edge has cost at most B.

As in the previous section, we define the notion of state for a vertex $v \in \texttt{red}(\mathcal{T})$. Let v be a node at reduced depth k in $\texttt{red}(\mathcal{T})$. Let $S_1, \dots, S_k$ be the segments encountered as we go from the root to v in $\mathcal{T}$. If we were to proceed as in the previous section, we will store a budget $B(S_i,\tau)$ For each cell (S_i,τ), $i = 1, \dots, k$, $\tau \in [\tau_{\min}, \tau_{\max}]$. This will lead to a very large number of possibilities (even if assume that for "nearby" cells, the budgets are not very different). Somewhat surprisingly, we show that it is enough to store this information at a small number of cells (in fact, linear in number of density classes and H).

To formalize this notion, we say that a sequence $\mathcal{C}_v =$

[1]For technical reasons, we will allow τ to lie in the range $[\tau_{\min}, \tau_{\max}+1]$

```
Fill DP Table :

  Input: A node $v \in \texttt{red}(\mathcal{T})$ at reduced depth $k$, and a state sequence $\Lambda_v = (B_1,...,B_k)$.
    0. If $v$ is a leaf node, set $D[v,\Lambda_v]$ to 0, and exit.
    1. Let $S_1,...,S_k$ be the segments visited while going from the root to $v$ in $\mathcal{T}$.
    2. Initialize $G(v) \leftarrow \emptyset$.
    3. For $i = 1,...,k$
         (i) Let $G_i(v)$ be the edges returned by GreedySelect($S_i$,$B_i$).
         (ii) $G(v) \leftarrow G(v) \cup G_i(v)$.
    4. Let $w_1,w_2$ be the two children of $v$ in $\texttt{red}(\mathcal{T})$ and
           the corresponding segments be $S^1_{k+1},S^2_{k+1}$.
    5. Initialize $M_1,M_2$ to $\infty$.
    6. For $r = 1,2$ (go to each of the two children and solve the subproblems)
         (i) For each extension $\Gamma_{w_r} = (B_1,...,B_k,B^r_{k+1})$ of $\Gamma_v$ do
               (a) Let $G_{k+1}(w_r)$ be the edges returned by GreedySelect($S^r_{k+1}$,$B^r_{k+1}$).
               (b) If any path in $\mathcal{P}_v$ ending in the segment $S^r_{k+1}$ is not satisfied by $G(v) \cup G_{k+1}(w_r)$
                       exit this loop
               (c) $M_r \leftarrow \min(M_r,\text{cost of } G_{k+1}(w_r) + D[w_r,\Gamma_{w_r}])$.
    7. $D[v,\Lambda_v] \leftarrow M_1 + M_2$.
```

Figure 6. Filling a table entry $D[v,\Lambda_v]$ in the dynamic program.

```
Algorithm GreedySelect:

  Input: A cell $(S,\tau)$ and a budget $B$.
  1. Initialize a set $G$ to emptyset.
  2. Let $S(\tau)$ be the edges in $S$ of density class $\tau$ and cost at most $B$.
  3. Arrange the edges in $S(\tau)$ from top to bottom order.
  4. Keep adding these edges to $G$ in this order till their total cost exceeds $2B$.
  5. Repeat Step 4 with the edges in $S(\tau)$ arranged in bottom to top order.
  6. Output $G$.
```

Figure 7. Algorithm GreedySelect for selecting edges in a segment S of density class τ with a budget B.

$\sigma_1,\dots,\sigma_\ell$ of cells is a *valid cell sequence* at v if the following conditions are satisfied : (i) the first cell σ_1 is $(S_k,\tau_{\max})$, (ii) the last cell is of the form (S_1,τ) for some density class τ, and (iii) if $\sigma = (S_i,\tau)$ is a cell in this sequence, then the next cell is either $(S_i,\tau-1)$ or $(S_{i-1},\tau+1)$. To visualize this definition, we arrange the cells (S_i,τ) in the form of a table shown in Figure 8. For each segment S_i, we draw a column in the table with one entry for each cell (S_i,τ), with τ increasing as we go up. Further as we go right, we shift these columns one step down. So row τ of this table will correspond to cells $(S_k,\tau),(S_{k-1},\tau+1),(S_{k-2},\tau+2)$ and so on. With this picture in mind, a a valid sequence of cells starts from the top left and at each step it either goes one step down or one step right. Note that for such a sequence $\mathcal{C}_v$ and a segment S_i, the cells (S_i,τ) which appear in $\mathcal{C}_v$ are given by $(S_i,\tau_1),(S_i,\tau_1+1),...,(S_i,\tau_2)$ for some $\tau_1 \leq \tau_2$. We say that the cells (S_i,τ), $\tau < \tau_1$, lie below the sequence $\mathcal{C}_v$, and the cells $(S_i,\tau),\tau > \tau_2$ lie above this sequence (e.g., in Figure 8, the cell $(S_2,6)$ lies above the shown cell sequence, and $(S_4,2)$ lies below it).

Besides a valid cell sequence, we need to define two more sequences for the vertex v:

- Valid Segment Budget Sequence: This is similar to the sequence defined in Section V-A2. This is a sequence $\Lambda^{\texttt{seg}}_v := (B^{\texttt{seg}}_1, \dots, B^{\texttt{seg}}_k)$, where $B^{\texttt{seg}}_i$ corresponds to the segment S_i. As before, each of these quantities is a power of 2 and lies in the range $[1,2n]$. Further, for any i, the ratio $B^{\texttt{seg}}_i/B^{\texttt{seg}}_{i+1}$ lies in the range [1/8,8].
- Valid Cell Budget Sequence: Corresponding to the valid cell sequence $\mathcal{C}_v = \sigma_1,\dots,\sigma_\ell$ and valid budget sequence $(B^{\texttt{seg}}_1, \dots, B^{\texttt{seg}}_k)$, we have a sequence $\Lambda^{\texttt{cell}}_v := (B^{\texttt{cell}}_1,...,B^{\texttt{cell}}_\ell)$, where $B^{\texttt{cell}}_j$ corresponds to the cell σ_j. Each of the quantities $B^{\texttt{cell}}_j$ lies in the range $[1,2n]$. Further. the ratio

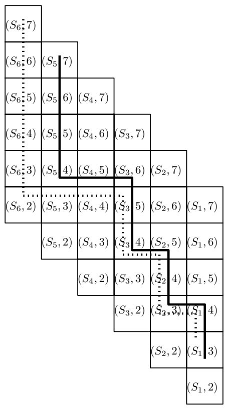
$(S_6,7)$
$(S_6,6)$ $(S_5,7)$
$(S_6,5)$ $(S_5,6)$ $(S_4,7)$
$(S_6,4)$ $(S_5,5)$ $(S_4,6)$ $(S_3,7)$
$(S_6,3)$ $(S_5,4)$ $(S_4,5)$ $(S_3,6)$ $(S_2,7)$
$(S_6,2)$ $(S_5,3)$ $(S_4,4)$ $(S_3,5)$ $(S_2,6)$ $(S_1,7)$
$(S_5,2)$ $(S_4,3)$ $(S_3,4)$ $(S_2,5)$ $(S_1,6)$
$(S_4,2)$ $(S_3,3)$ $(S_2,4)$ $(S_1,5)$
$(S_3,2)$ $(S_2,3)$ $(S_1,4)$
$(S_2,2)$ $(S_1,3)$
$(S_1,2)$

Figure 8. w is a vertex at reduced depth 6 and v is the parent of v in $\texttt{red}(\mathcal{T})$. The segments above w are labelled S_1,...,S_6 (starting from the root downwards). The cells are arranged in a tabular fashion as shown – the density classes lie in the range {2,3,...,7}. The solid line shows a valid cell sequence for v. The dotted line shows a valid cell sequence for w which is also an extension of the cell sequence for v – note that once the dotted line meets the solid line (in the cell (S_3,5), it stays with it till the end.

$B_j^{\texttt{cell}}/B_{j+1}^{\texttt{cell}}$ lies in the range [1/8,8].

Intuitively, $B_i^{\texttt{seg}}$ is supposed to capture the cost of edges picked by the optimal solution in S_i, whereas $B_j^{\texttt{cell}}$, where $\sigma_j = (S_i, \tau)$, captures the cost of the density class τ edges in S_i which get selected by the optimal solution. A *valid state* $\texttt{State}(v)$ at the vertex v is given by the triplet $(\mathcal{C}_v, \Lambda_v^{\texttt{seg}}, \Lambda_v^{\texttt{cell}})$ which in addition satisfies the following properties:

(i) For a cell $\sigma_j = (S_i, \tau)$ in $\mathcal{C}_v$, the quantity $B_j^{\texttt{cell}} \leq B_i^{\texttt{seg}}$. Again, the intuition is clear – the first quantity corresponds to cost of density class τ edges in S_i, whereas the latter denotes the cost of all the edges in S_i (which are selected by the optimal solution).[2]

Informally, the idea behind these definitions is the following – for each cell σ_j in $\mathcal{C}_v$, we are given the corresponding budget $B_j^{\texttt{cell}}$. We use this budget and Algorithm `GreedySelect` to select edges corresponding to this cell. For cells $\sigma = (S_i,\tau)$ which lie above $\mathcal{C}_v$, we do not have to use any edge of density class τ from S_i. Note that this does not mean that our algorithm will not pick any such edge, it is just that for the sub-problem defined by the paths in $\mathcal{P}_v$ and the state at v, we will not use any such edge (for covering a path in $\mathcal{P}_v$). For cells $\sigma = (S_i,\tau)$ which lie below $\mathcal{C}_v$, we pick all edges of density class τ and cost at most $B_i^{\texttt{seg}}$ from S_i. Thus, we can specify the subset of selected edges from $S_1, \ldots , S_k$ (for the purpose of covering paths in $\mathcal{P}_v$) by specifying these sequences only. The non-trivial fact is to show that maintaining such a small state (i.e., the three valid sequences) suffices to capture all scenarios. The algorithm for picking the edges for a specific segment S is shown in Figure 9. Note one subtlety – for the density class τ_1 (in Step 4), we use budget $B_i^{\texttt{seg}}$ instead of the corresponding cell budget.

[2]During the analysis, $B_i^{\texttt{seg}}$ will be the *maximum* over all density classes τ of the density class τ edges selected by the optimal solution from this segment. But this inequality will still hold.

Before specifying the DP table, we need to show what it means for a state to be an *extension* of another state. Let w be a child of v in $\texttt{red}(\mathcal{T})$, and let S_{k+1} be the corresponding segment joining v and w. Given states $\texttt{State}(v) := (\mathcal{C}_v, \Lambda_v^{\texttt{Seg}}, \Lambda_v^{\texttt{cell}})$ and $\texttt{State}(w) := (\mathcal{C}_w, \Lambda_w^{\texttt{Seg}}, \Lambda_w^{\texttt{cell}})$, we say that $\texttt{State}(w)$ is an extension of $\texttt{State}(v)$ if the following conditions are satisfied:

- If $\Lambda_v^{\texttt{seg}} = (B_1^{\texttt{seg}}, \ldots , B_k^{\texttt{seg}})$ and $\Lambda_w^{\texttt{seg}} = (B_1^{\texttt{seg}'}, \ldots, B_{k+1}^{\texttt{seg}'})$, then $B_i^{\texttt{seg}} = B_i^{\texttt{seg}'}$ for $i = 1, \ldots, k$. In other words, the two sequences agree on segments $S_1, \ldots, S_k$.
- Recall that the first cell of $\mathcal{C}_w$ is $(S_{k+1}, \tau_{\max})$. Let τ_1 be the smallest τ such that the cell (S_{k+1}, τ_1) appears in $\mathcal{C}_w$. Then the cells succeeding (S_{k+1}, τ_1) in $\mathcal{C}_w$ must be of the form $(S_k, \tau_1+1), (S_{k-1}, \tau_1+2), \ldots,$ till we reach a cell which belongs to $\mathcal{C}_v$ (or we reach a cell for the segment S_1). After this the remaining cells in $\mathcal{C}_w$ are the ones appearing in $\mathcal{C}_v$. Pictorially (see Figure 8), the sequence for $\mathcal{C}_w$ starts from the top left, keeps going down till (S_{k+1}, τ_1), and then keeps moving right till it hits $\mathcal{C}_v$. After this, it merges with $\mathcal{C}_v$.
- The sequences $\Lambda_v^{\texttt{cell}}$ and $\Lambda_w^{\texttt{cell}}$ agree on cells which belong to both $\mathcal{C}_v$ and $\mathcal{C}_w$ (note that the cells common to both will be a suffix of both the sequences).

Having defined the notion of extension, the algorithm for filling the DP table for $D[v, \texttt{State}(v)]$ is identical to the one in Figure 6. The details are given in Figure 10. This completes the description of the algorithm.

For the full version of the paper with proofs, please refer to the link https://arxiv.org/abs/1802.07439

VI. Discussion

We give the first pseudo-polynomial time constant factor approximation algorithm for the weighted flow-time problem on a single machine. The algorithm can be made to run in time polynomial in n and W as well, where W is the ratio of the maximum to the minimum weight. The rough idea is as follows. We have already assumed that the costs of the job segments are polynomially bounded (this is without loss of generality). Since the cost of a job segment is its weight times its length,

```
Algorithm SelectSegment:
  Input: A vertex $v \in \mathtt{red}(\mathcal{T})$, $\mathtt{State}(v) := (\mathcal{C}_v, \Lambda_v^{\mathtt{Seg}}, \Lambda_v^{\mathtt{cell}})$, a segment $S_i$ lying above $v$.
    1. Initialize a set $G$ to emptyset.
    2. Let $(S_i,\tau_1),(S_i,\tau_1+1),\ldots,(S_i,\tau_2)$ be the cells in $\mathcal{C}_v$ corresponding to the segment $S_i$.
    3. For $\tau=\tau_1+1,\ldots,\tau_2$ do
          (i) Add to $G$ the edges returned by GreedySelect($(S_i,\tau),B_j^{\mathtt{cell}}$),
                where $j$ is the index of $(S_i,\tau)$ in $\mathcal{C}_v$.
    4. Add to $G$ the edges returned by GreedySelect($(S_i,\tau_1),B_i^{\mathtt{seg}}$).
    5. Add to $G$ all edges $e \in S_i$ of density class strictly less than $\tau$ and for which $c_e \le B_i^{\mathtt{seg}}$.
    6. Return $G$.
```

Figure 9. Algorithm SelectSegment for selecting edges in a segment S as dictated by the state at v. The notations $B^{\mathtt{seg}}$ and $B^{\mathtt{cell}}$ are as explained in the text.

```
Fill DP Table :
  Input: A node $v \in \mathtt{red}(\mathcal{T})$ at reduced depth $k$, $\mathtt{State}(v) = (\mathcal{C}_v, \Lambda_v^{\mathtt{Seg}}, \Lambda_v^{\mathtt{cell}})$.
    0. If $v$ is a leaf node, set $D[v,\mathtt{State}(v)]$ to 0, and exit.
    1. Let $S_1,\ldots,S_k$ be the segments visited while going from the root to $v$ in $\mathcal{T}$.
    2. Initialize $G(v) \leftarrow \emptyset$.
    3. For $i=1,\ldots,k$
       (i) Let $G_i(v)$ be the edges returned by Algorithm SelectSegment($v,S_i,\mathtt{State}(v)$).
       (ii) $G(v) \leftarrow G(v) \cup G_i(v)$.
    4. Let $w_1,w_2$ be the two children of $v$ in $\mathtt{red}(\mathcal{T})$ and
          the corresponding segments be $S_{k+1}^1,S_{k+1}^2$.
    5. Initialize $M_1,M_2$ to $\infty$.
    6. For $r=1,2$ (go to each of the two children and solve the subproblems)
       (i) For each extension $\mathtt{State}(w_r)$ of $\mathtt{State}(v)$ do
             (a) Let $G_{k+1}(w_r)$ be the edges returned by SelectSegment($w_r,S_{k+1}^r,\mathtt{State}(w_r)$).
             (b) If any path in $\mathcal{P}_v$ ending in the segment $S_{k+1}^r$ is not satisfied by $G(v) \cup G_{k+1}(w_r)$
                   exit this loop
             (c) $M_r \leftarrow \min(M_r, \text{cost of } G_{k+1}(w_r) + D[w_r,\mathtt{State}(w_r)])$.
    7. $D[v,\Lambda_v] \leftarrow M_1+M_2$.
```

Figure 10. Filling a table entry $D[v,\mathtt{State}(v)]$ in the dynamic program.

it follows that the lengths of the job segments are also polynomially bounded, say in the range $[1,n^c]$. Now we ignore all jobs of size less than $1/n^2$, and solve the remaining problem using our algorithm (where P will be polynomially bounded). Now, we introduce these left out jobs, and show that increase in weighted flow-time will be small. Further, the algorithm also extends to the problem of minimizing ℓ_p norm of weighted flow-times. We can do this by changing the objective function in (IP2) to $(\sum_j \sum_s (w(j,S))^p y(j,S))^{1/p}$ and showing that this is within a constant factor of the optimum value. The instance of Demand MultiCut in the reduction remains exactly the same, except that the weights of the nodes are now $w(j,S)^p$. We leave the problem of obtaining a truly polynomial time constant factor approximation algorithm as open.

Acknowledgement

The authors would like to thank Sungjin Im for noting the extension to ℓ_p norm of weighted flow-times.

References

[1] Nikhil Bansal and Kedar Dhamdhere. Minimizing weighted flow time. *ACM Trans. Algorithms*, 3(4):39, 2007.

[2] Nikhil Bansal, Ravishankar Krishnaswamy, and Barna Saha. On capacitated set cover problems. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 14th International Workshop, APPROX 2011, and 15th International Workshop, RANDOM 2011, Princeton, NJ, USA, August 17-19, 2011. Proceedings*, pages 38–49, 2011.

[3] Nikhil Bansal and Kirk Pruhs. Server scheduling in the weighted lp norm. In *LATIN 2004: Theoretical Informatics, 6th Latin American Symposium, Buenos Aires, Argentina, April 5-8, 2004, Proceedings*, pages 434–443, 2004.

[4] Nikhil Bansal and Kirk Pruhs. The geometry of scheduling. *SIAM J. Comput.*, 43(5):1684–1698, 2014.

[5] Amotz Bar-Noy, Reuven Bar-Yehuda, Ari Freund, Joseph Naor, and Baruch Schieber. A unified approach to approximating resource allocation and scheduling. *J. ACM*, 48(5):1069–1090, 2001.

[6] Michael A. Bender, S. Muthukrishnan, and Rajmohan Rajaraman. Approximation algorithms for average stretch scheduling. *J. Scheduling*, 7(3):195–222, 2004.

[7] Deeparnab Chakrabarty, Elyot Grant, and Jochen Könemann. On column-restricted and priority covering integer programs. In *Integer Programming and Combinatorial Optimization, 14th International Conference, IPCO 2010, Lausanne, Switzerland, June 9-11, 2010. Proceedings*, pages 355–368, 2010.

[8] Timothy M. Chan, Elyot Grant, Jochen Könemann, and Malcolm Sharpe. Weighted capacitated, priority, and geometric set cover via improved quasi-uniform sampling. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 1576–1585, 2012.

[9] Chandra Chekuri and Sanjeev Khanna. Approximation schemes for preemptive weighted flow time. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 297–305, 2002.

[10] Chandra Chekuri, Sanjeev Khanna, and An Zhu. Algorithms for minimizing weighted flow time. In *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 84–93, 2001.

[11] Naveen Garg, Vijay V. Vazirani, and Mihalis Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica*, 18(1):3–20, 1997.

[12] Sungjin Im and Benjamin Moseley. Fair scheduling via iterative quasi-uniform sampling. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2601–2615, 2017.

[13] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *J. ACM*, 47(4):617–643, 2000.

[14] Kasturi R. Varadarajan. Weighted geometric set cover via quasi-uniform sampling. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 641–648, 2010.