SIAM J. COMPUT.
Vol. 52, No. 6, pp. FOCS18-158–FOCS18-188
© 2020 Society for Industrial and Applied Mathematics

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

CONSTANT FACTOR APPROXIMATION ALGORITHM FOR WEIGHTED FLOW-TIME ON A SINGLE MACHINE IN PSEUDOPOLYNOMIAL TIME*

JATIN BATRA[†], NAVEEN GARG[‡], AND AMIT KUMAR[‡]

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 multicut problem on trees, which we call the `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 the dynamic programming table.

Key words. dynamic programming, scheduling, multicut

AMS subject classification. 68W25

DOI. 10.1137/19M1244512

1. 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 n 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 shortest remaining processing time policy gives an optimal algorithm for this objective.

In the weighted version of this problem, each job j has a weight w_j, and we would like to minimize the weighted sum of flow-time of jobs. 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 a polynomial time approximation scheme (PTAS)) for it. In a breakthrough result, Bansal and Pruhs [6] gave a $O(\log \log P)$ approximation for the problem in polynomial time where P denotes the ratio of the largest to the smallest processing time of a job in the instance.

In this paper, we make substantial progress towards this problem by giving the first constant factor approximation algorithm for this problem in pseudopolynomial time. In fact, given a $p \geq 1$, our result holds for the ℓ_p norm of weighted flow-times. More formally, we prove the following result.

*Received by the editors February 19, 2019; accepted for publication (in revised form) July 2, 2020; published electronically September 29, 2020.
https://doi.org/10.1137/19M1244512
[†]CWI, Amsterdam, the Netherlands (jatinbatra50@gmail.com).
[‡]Department of Computer Science and Engineering, IIT Delhi, New Delhi, India (naveen@cse.iitd.ac.in, amitk@cse.iitd.ac.in).

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

THEOREM 1.1. *There is a constant factor approximation algorithm for* `WtdFlowTime` *where the running time of the algorithm is polynomial in n and P.*

In fact, our approximation guarantee extends to ℓ_p norm of weighted flow-times.

We obtain this result by reducing `WtdFlowTime` to a generalization of the multicut problem on trees, which we call `Demand MultiCut`. The `Demand MultiCut` problem is a natural generalization of the multicut 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. Recall that the usual multicut problem requires us to find a minimum cost subset of edges whose removal disconnects every terminal pair. Hence we see that when all demands and sizes are 1, `Demand MultiCut` is the multicut problem.

The natural integer program for this problem has the property that all nonzero entries in any column of the constraint matrix are the same. Such integer programs, called *column restricted covering integer programs*, were studied by Chakrabarty, Grant, and Könemann [9]. 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 linear programming (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. The priority version is the setting 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 round the LP relaxation of the priority version. This is similar to the situation faced by Bansal and Pruhs [6], where they need to round the priority version of a geometric set cover problem. They appeal to the notion of shallow cell complexity [10] 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) [10], 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.

1.1. Related work. There has been a lot of work on the `WtdFlowTime` problem on a single machine. Bansal and Pruhs [6] made significant progress towards this problem setting 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 [7], implies a constant factor approximation algorithm for this setting when all release dates are 0. Chekuri and Khanna [11] gave a quasi-PTAS for this problem if P, W are polynomially bounded, 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 [8, 11]. The problem of minimizing (unweighted) ℓ_p norm of flow-times was studied by Im and Moseley [15] who gave a constant factor approximation in polynomial time.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Only strong NP-hardness is known for `WtdFlowTime` [18]. Note, however, that if preemption is not allowed, then the problem becomes hard to approximate within $n^{\frac{1}{2}-\epsilon}$ for any $\epsilon > 0$ as shown by Kellerer, Tautenhahn, and Woeginger [17].

In the online setting, Bansal and Dhamdhere [3] gave an $O(\log W)$-competitive algorithm for `WtdFlowTime`, where W is the ratio of the maximum to the minimum weight of a job. They also gave a semionline (where the algorithm needs to know the parameters P and W in advance) $O(\log(nP))$-competitive algorithm, where P is the ratio of the largest to the smallest processing time of a job. Chekuri, Khanna, and Zhu [12] gave a semionline $O(\log^2 P)$-competitive algorithm. These results were recently improved by Azar and Touitou [1] who gave a $\min(\log W, \log P, \log D)$ competitive algorithm where D is the maximum-to-minimum density ratio of jobs, density of a job j being defined as w_j/p_j. On the lower bound side, Bansal and Chan [2] showed that there is no constant competitive online algorithm for `WtdFlowTime`.

In order to understand the online setting better in the presence of lower bound results such as [2], the speed augmentation model was introduced by Kalyanasundaram and Pruhs [16], in which the algorithm is given $(1+\varepsilon)$-times extra speed than the optimal algorithm. In this model, Bansal and Pruhs [5] showed that the highest density first algorithm is $O(1)$-competitive for weighted ℓ_p norms of flow-time for all values of $p \geq 1$.

The multicut problem on trees is known to be NP-hard, and a 2-approximation algorithm was given by Garg, Vazirani, and Yannakakis [14]. As mentioned earlier, Chakrabarty, Grant, and Könemann [9] gave a systematic study of column restricted covering integer programs (see also [4] for follow-up results). The notion of shallow cell complexity for 0-1 covering integer programs was formalized by Chan et al. [10], where they relied on and generalized the techniques of Vardarajan [19].

Subsequent to our work, Feige, Kulkarni, and Li [13] have shown that any pseudo-polynomial time algorithm for `WtdFlowTime` can be transformed into a polynomial time algorithm at $O(1)$ loss in the approximation ratio. Together with our result, this implies a polynomial time constant factor approximation for `WtdFlowTime`.

2. 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 without loss of generality 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 have exactly $T = \sum_j p_j$ slots where jobs are being processed. 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 multicut 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.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Overview of paper. In section 2.1, we describe a well-known integer program (IP) for `WtdFlowTime`. This IP has variables $x_{j,t}$ for every job j and time $t \geq r_j$. Intuitively, we think of setting $x_{j,t}$ to be 1 if j completes processing after time t. The IP has several covering constraints. However, the IP must also include the noncovering constraints specifying that $x_{j,t} \leq x_{j,t-1}$ for all $t \geq r_j$, which creates issues for our approach. To get around this, we first proceed as in [6] and 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 (up to a factor 2 loss) round the job's flow-time up to 2^{i+1} 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. Our novel idea now is to ensure that these intervals are laminar for different jobs at only $O(1)$ loss in our new IP (IP3). From here, the reduction to the `Demand MultiCut` problem follows (see section 4 for details).

In section 5, 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 restated as—any root to leaf path has at most $O(\log(nP))$ segments. We give a dynamic programming algorithm for such instances. To compute the entry corresponding to a vertex v in the tree, our algorithm looks at the table entries corresponding to the subtree rooted below v. However, to be able to correctly compute such a table entry, we must also keep track of the "state" of the ancestor edges of v, where the state means the subset of the ancestor edges selected by the algorithm. Doing this naively requires too much book-keeping. Therefore, we use the following two ideas to reduce the size of this required extra 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 neighboring segments. (ii) If we could spend twice the amount spent by the algorithm on a segment S and select low density edges (where density of an edge is the ratio of its cost and size), we could ignore the edges in a segment S' lying above S in the tree at $O(1)$ loss. The heart of the technique is the second idea above; due to this idea we no longer need to remember the amounts spent by the algorithm on high density edges on segments lying above. Further, the selection of additional low density edges from this segment also ends up reducing the amount of information needed to remember which low density edges were selected. These ideas lead us to define a low information state which lends itself naturally to a an optimal substructure, which we exploit to produce an efficient dynamic program.

2.1. An exact IP. We describe an IP for the `WtdFlowTime` problem. This is well known (see, e.g., [6]), but we give details for sake of completeness. We will have binary variables $x_{j,t}$ for every job j and time t, where $r_j \leq t \leq T$. This variable is meant to be 1 iff j is *alive* at time t, i.e., its completion time is at least t. In the objective function, we will minimize the pth power of the ℓ_p norm of weighted flow-times which is $\sum_j (\sum_{t \in [r_j, T]} w_j x_{j,t})^p$. We now specify the constraints of the IP. Consider a time interval $I = [s, t]$, where $0 \leq s \leq t \leq T$, and s and t are integers. Let

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

$l(I)$ 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 IP (IP1):

$$\min \sum_j \sum_{t \in [r_j, T]} w_j x_{j,t}, \tag{2.1}$$

$$\sum_{j \in J(I)} x_{j,t} p_j \geq p(J(I)) - l(I) \qquad \text{for all intervals } I = [s,t], 0 \leq s \leq t \leq T, \tag{2.2}$$

$$x_{j,t} \leq x_{j,t-1} \qquad \text{for all jobs } j \text{ and time } t,\ r_j < t \leq T, \tag{2.3}$$

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

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 IP, there is a corresponding schedule of the same cost.

THEOREM 2.1. *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 $x_{j,t}$.*

Proof. We show how to build such a schedule. The integral solution x gives us deadlines for each job. For a job j, define d_j as one plus the last time t such that $x_{j,t} = 1$. Note that $x_{j,t} = 1$ for every $t \in [r_j, d_j)$. We would like to find a schedule which completes each job by time d_j: if such a schedule exists, then the weighted flow-time of a job j will be at most $\sum_{t \geq r_j} w_j x_{j,t}$, which is what we want.

We begin by observing a simple property of a feasible solution to the IP.

CLAIM 2.2. *Consider an interval $I = [s,t]$, $0 \leq s \leq t \leq T$. Let J' be a subset of $J(I)$ such that $p(J') > l(I)$. If x is a feasible solution to* IP1, *then there must exist a job $j \in J'$ such that $x_{j,t} = 1$.*

Proof. Suppose not. Then the left-hand side (LHS) of constraint (2.2) for I would be at most $p(J(I) \setminus J')$, whereas the right-hand side (RHS) would be $p(J') + p(J(I) \setminus J') - l(I) > p(J(I) \setminus J')$, a contradiction. ☐

It is natural to use the earliest deadline first rule to find the required schedule. We build the schedule from time $t = 0$ onwards. At any time t, we say that a job j is *alive* if $r_j \leq t$, and j has not been completely processed by time t. Starting from time $t = 0$, we process the alive job with earliest deadline d_j during $[t, t+1]$. We need to show that every job will complete before its deadline. Suppose not. Let j be the job with the earliest deadline which is not able to finish by d_j. Let t be first time before d_j such that the algorithm processes a job whose deadline is more than d_j during $[t-1, t]$, or it is idle during this time slot (if there is no such time slot, it must have busy from time 0 onwards, and so set t to 0). The algorithm processes jobs whose deadline is at most d_j during $[t, d_j]$—call these jobs J'. We claim that jobs in J' were released after t—indeed, if such a job was released before time t, it would have been alive at time $t-1$ (since it gets processed after time t). Further, its deadline is at most d_j, and so the algorithm should not be processing a job whose deadline is more than d_j during $[t-1, t]$ (or being idle). But now, consider the interval $I = [t, d_j]$. Observe that $l(I) < p(J')$—indeed, $j \in J'$, and it is not completely processed during

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

I, but the algorithm processes jobs from J' only during I. Claim 2.2 now implies that there must be a job j' in J' for which $x_{j',d_j} = 1$—but then the deadline of j' is more than d_j, a contradiction. □

3. A different IP. We now write a weaker IP, but it has more structure in it. This has two goals—to get rid of the monotonicity constraint 2.3 and to obtain more structure on the sets of jobs involved in the knapsack constraint 2.2. 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 nonnegative integers i and s (we shall use segments to denote such intervals to avoid any confusion with intervals used in the IP). 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 $[r_j, T]$.

Algorithm FormSegments(j)
1. Initialize $t \leftarrow r_j$.
2. For $s = 0, 1, 2, \ldots,$
    (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 $\texttt{Seg}(j)$
        update $t \leftarrow t + 2^{s+1}$.
    (ii) Else add the segment (from the set $\mathcal{S}_s$) $[t, , t+2^s]$ to $\texttt{Seg}(j)$.
        update $t \leftarrow t + 2^s$.

FIG. 1. *Forming* $\texttt{Seg}(j)$.

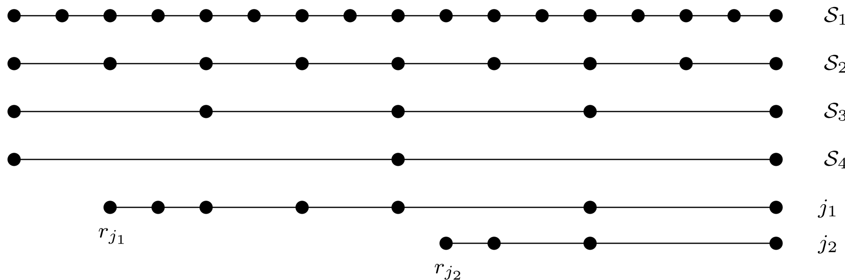

FIG. 2. *The dyadic segments* $\mathcal{S}_1, \ldots, \mathcal{S}_4$ *and the corresponding* $\texttt{Seg}(j_1), \texttt{Seg}(j_2)$ *for two jobs* j_1, j_2.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

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 3.1. *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 $s \geq s'$.*

Proof. We prove this by induction on t. When $t = r_{j'}$, this is trivially true because s' would be 0. Suppose it is true for some $t \geq r_{j'}$. Let (j, S) and (j', S') be the job segments containing t. Suppose $S \in \mathcal{S}_s, S' \in \mathcal{S}_{s'}$. By the induction hypothesis, we know that $s \geq s'$. Let $(j', \tilde{S}')$ be the job segment containing $t+1$, and let $\tilde{S}' \in \mathcal{S}_{\tilde{s}'}$ (S' could be the same as $\tilde{S}'$). We know that $\tilde{s}' \leq s' + 1$. Therefore, the only interesting case is $s = s'$ and $\tilde{s}' = s' + 1$. Since $s = s'$, the two segments S and S' must be the same (because all segments in $\mathcal{S}_s$ are mutually disjoint). Since $t \in S, t+1 \notin S$, it must be that $S = [l, t+1]$ for some l. The algorithm for constructing $\texttt{Seg}(j')$ adds a segment from $\mathcal{S}_{s'+1}$ after adding S' to $\texttt{Seg}(j')$. Therefore $t+1$ must be a multiple of $2^{s'+1}$. What does the algorithm for constructing $\texttt{Seg}(j)$ do after adding S to $\texttt{Seg}(j)$? If it adds a segment from $\mathcal{S}_{s+1}$, then we are done again. Suppose it adds a segment from $\mathcal{S}_s$. The right end-point of this segment would be $(t+1) + 2^s$. After adding this segment, the algorithm would add a segment from $\mathcal{S}_{s+1}$ (as it cannot add more than 2 segments from $\mathcal{S}_s$ to $\texttt{Seg}(j)$). But this can only happen if $(t+1) + 2^s$ is a multiple of 2^{s+1}—this is not true because $(t+1)$ is a multiple of 2^{s+1}. Thus we get a contradiction, and so the next segment (after S) in $\texttt{Seg}(j)$ must come from $\mathcal{S}_{s+1}$ as well. ∎

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 IP (IP2):

$$\min \sum_j \sum_s w(j, S) y(j, S), \tag{3.1}$$

$$\sum_{(j,S): j \in J(I), t \in (j,S)} y(j, S) p_j \geq p(J(I)) - l(I) \qquad \text{for all } I = [s, t], 0 \leq s \leq t \leq T, \tag{3.2}$$

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

Observe that for any interval I, the constraint (3.2) 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 (2.3), 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 IPs.

LEMMA 3.2. *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.*

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Proof. Suppose we are given a solution x for IP1. For every job j, let d_j be the highest t for which $x_{j,t} = 1$. Let $S_1, S_2, \ldots$ be the segments in $\texttt{Seg}(j)$ indexed by the order in which they were added. Let S_{i_j} be the segment in $\texttt{Seg}(j)$ which contains d_j. Then we set $y(j, S_i)$ to 1 for all $i \leq i_j$ and $y(j, S_i)$ to 0 for all $i > i_j$. This defines the solution y. First we observe that y is feasible for IP2. Indeed, consider an interval $I = [s, t]$. If $x_{j,t} = 1$ and $j \in J(I)$, then we do have $y(j, S) = 1$ for the job segment (j, S) containing t. Therefore, the LHS of constraints (2.2) and (3.2) for I are the same. Also, observe that

$$\sum_{S \in \texttt{Seg}(j)} y(j, S) w(j, S) = \sum_{i=1}^{i_j} w_j \cdot l(S_i) \leq w_j 4 l(S_{i_j}),$$

where the last inequality follows from the fact that there are at most two segments from any particular set $\mathcal{S}_s$ in $\texttt{Seg}(j)$, and so the length of every alternate segment in $\texttt{Seg}(j)$ increases exponentially. So, $\sum_{i=1}^{i_j} l(s_i) \leq 2(l(S_{i_j}) + l(S_{i_j-2}) + l(S_{i_j-4}) + \cdots) \leq 4 \cdot l(S_{i_j})$. Finally observe that $l(S_{i_j}) \leq 2(d_j - r_j)$. Indeed, the length of $S_{i_{j-1}}$ is at least half of that of S_{i_j}. So,

$$l(S_{i_j}) \leq 2 l(S_{i_j-1}) \leq 2(d_j - r_j).$$

Thus, the total contribution to the cost of y from job segments corresponding to j is at most $8w_j(d_j - r_j) = 8w_j \sum_{t \geq r_j} x_{j,t}$.

This proves the first statement in the lemma.

Now we prove the second statement. Let y be a solution to IP2. For each job j, let S_{i_j} be the last job segment in $\texttt{Seg}(j) = \{S_1, S_2, \ldots\}$ for which $y(j, S)$ is 1. We set $x_{j,t}$ to 1 for every $t \leq d_j$, where d_j is the right end-point of S_{i_j}, and 0 for $t > d_j$. It is again easy to check that x is a feasible solution to IP1. For a job j the contribution of j towards the cost of x is

$$w_j(d_j - r_j) = w_j \cdot \sum_{i=1}^{i_j} l(S_i) \leq 4 w_j \cdot l(S_{i_j}) \leq 4 \cdot \sum_{(j,S) \in \texttt{Seg}(j)} w(j, S) y(j, S). \qquad \square$$

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 (3.2) 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 IP.

4. 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 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 edges V' such that for every path $P \in \mathcal{P}$, the set of edges 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`.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

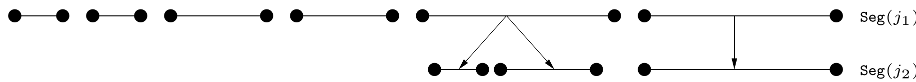

Fig. 3. *The forest formed by the segments of jobs j_1 and j_2. The forest consists of six trees.*

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 $\mathcal{T}$ (see Figure 3). 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 3.1. 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 (3.2) 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 (3.2) corresponding to it. Let V_I be the vertices in $\mathcal{T}$ corresponding to the job segments appearing in the LHS of this constraint.

Lemma 4.1. *The vertices in V_I form a path in $\mathcal{T}$ from an ancestor to a descendant.*

Proof. Let $j_1, \ldots, j_k$ be the jobs which are released in I arranged according to $\prec_J$. Note that these will form a consecutive subsequence of the sequence obtained by arranging jobs according to $\prec_J$. Each of these jobs will have exactly one job segment (j_i, S_i) appearing on the LHS of this constraint (because for any such job j_i, the segments in $\texttt{Seg}(j_i)$ partition $[r_{j_i}, T]$). All these job segments contain t, and so these segment intersect. Now, by construction of $\mathcal{T}$, it follows that the parent of (j_i, S_i) in the tree $\mathcal{T}$ would be (j_{i-1}, S_{i-1}). This proves the claim. ∎

Let the vertices in V_I be $v_1, \ldots, v_k$ 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 $P_I = v_0, v_1, \ldots, v_k$ to $\mathcal{P}$—Lemma 4.1 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 (3.2) for the interval I. The following claim is now easy to check.

Claim 4.2. *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.*

Proof. Consider a solution to $\mathcal{I}$ consisting of a set of edges E. For each edge $e = (v_1, v_2) \in E$ where $v_2 = (j, S)$ is the child of v_1, we set $y(j, S) = 1$. For rest of the job segments (j, S), define $y(j, S)$ to be 0. Since the cost of such an edge e is equal to $w(j, S)$, it is easy to see that the two solutions have the same cost. Feasibility of IP2 also follows directly from the manner in which the paths in $\mathcal{P}$ are defined. ∎

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

This completes the reduction from `WtdFlowTime` to `Demand MultiCut`. This reduction is polynomial time because the 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`.

5. 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 $\texttt{red}(\mathcal{T})$ denote the tree obtained from $\mathcal{T}$ by shortcutting all nonroot degree 2 vertices (see Figure 4 for an example). There is a clear correspondence between the vertices of $\texttt{red}(\mathcal{T})$ and the nonroot vertices in $\mathcal{T}$ which do not have degree 2. We shall use the same notation for both these sets of vertices.

The reduced height of $\mathcal{T}$ is defined as the height of $\texttt{red}(\mathcal{T})$. In this section, we prove the following result. We say that a (rooted) tree is binary if every node has at most 2 children.

THEOREM 5.1. *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}.

Before we prove this theorem, let us see why it implies the main result in Theorem 1.1.

Proof of Theorem 1.1. Consider an instance $\mathcal{I} = (\mathcal{T}, \mathcal{P}, c, p, d)$ of `DemandMultiCut` obtained via reduction from an instance $\mathcal{I}'$ of `WtdFlowTime`. Let n' denote the number of jobs in $\mathcal{I}'$ and P denote the ratio of the largest to the smallest job size in this instance. We had argued in the previous section that n, the number of nodes in $\mathcal{T}$, is $O(n'\log P)$. We first perform some preprocessing on $\mathcal{T}$ such that the quantites $H, \rho_{\max}/\rho_{\min}$ do not become too large.

- Let $p_{\max}$ and $p_{\min}$ denote the maximum and the minimum size of a job in the instance $\mathcal{I}'$. Each edge in $\mathcal{T}$ corresponds to a job interval in the instance

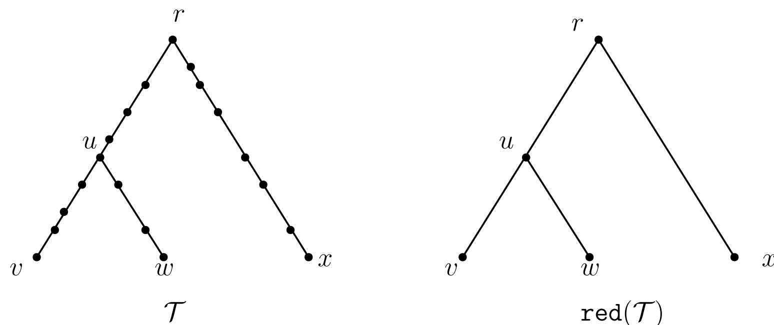

FIG. 4. *Tree $\mathcal{T}$ and the corresponding tree* $\texttt{red}(\mathcal{T})$. *Note that the vertices in* $\texttt{red}(\mathcal{T})$ *are also present in $\mathcal{T}$, and the segments in $\mathcal{T}$ correspond to edges in* $\texttt{red}(\mathcal{T})$. *The tree $\mathcal{T}$ has 4 segments,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

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

$\mathcal{I}$. We select all edges for which the corresponding job interval has length at most $p_{\min}$. Note that after selecting these edges, we will contract them in $\mathcal{T}$ and adjust the demands of paths in $\mathcal{P}$ accordingly. For a fixed job j, the total cost of such selected edges would be at most $4w_j p_{\min} \le 4w_j p_j$ (as in the proof of Lemma 3.2, the corresponding job intervals have lengths which are powers of 2, and there are at most two intervals of the same length). Note that the cost of any optimal solution for $\mathcal{I}'$ is at least $\sum_j w_j p_j$, and so we are incurring an extra cost of at most 4 times the cost of the optimal solution.

So we can assume that any edge in $\mathcal{T}$ corresponds to a job interval in $\mathcal{I}'$ whose length lies in the range $[p_{\min}, n' p_{\max}]$, because the length of the schedule is at most $n' p_{\max}$ (recall that we are assuming that there are no gaps in the schedule).

- Let $c_{\max}$ be the maximum cost of an edge selected by the optimal solution (we can cycle over all n possibilities for $c_{\max}$ and select the best solution obtained over all such solutions). We remove (i.e., contract) all edges of cost more than $c_{\max}$ and select all edges of cost at most $c_{\max}/n$ (i.e., contract them and adjust demands of paths going through them)—the cost of these selected edges will be at most a constant times the optimal cost. Therefore, we can assume that the costs of the edges lie in the range $[c_{\max}/n, c_{\max}]$. Therefore, the densities of the edges in $\mathcal{T}$ lie in the range $[\frac{c_{\max}}{n p_{\max}}, \frac{c_{\max}}{p_{\min}}]$.

Having performed the above steps, we now modify the tree $\mathcal{T}$ so that it becomes a binary tree. Recall that each vertex v in $\mathcal{T}$ corresponds to a dyadic interval S_v, and if w is a child of v, then S_w is contained in S_v (for the root vertex, we can assign it the dyadic interval $[0, T]$). Now, consider a vertex v with S_v of size 2^s, and suppose it has more than 2 children. Since the dyadic intervals for the children are mutually disjoint and contained in S_v, each of these will be of size at most 2^{s-1}. Let S_v^1 and S_v^2 be the two dyadic intervals of length 2^{s-1} contained in S_v. Consider S_v^1. Let $w_1, \ldots, w_k$ be the children of v for which the corresponding interval is contained in S_v^1. If $k > 1$, we create a new node w below v (with corresponding interval being S_v^1) and make $w_1, \ldots, w_k$ children of v. The cost and size of the edge (v, w) is 0. We proceed similarly for S_v^2. Thus, each node will now have at most 2 children. Note that we will blow up the number of vertices by a factor of 2.

We can now estimate the reduced height H of $\mathcal{T}$. Consider a root to leaf path in $\texttt{red}(\mathcal{T})$, and let the vertices in this path be $v_1, \ldots, v_k$. Let e_i denote the parent of v_i. Since each v_i has two children in $\mathcal{T}$, the job interval corresponding to e_i will be at least twice that for e_{i+1}. From the first preprocessing step above, it follows that the length of this path is bounded by $\log(n'P)$, where P denotes $p_{\max}/p_{min}$. Thus, H is $O(\log(n'P))$. It now follows from Theorem 5.1 that we can get a constant factor approximation algorithm for the instance $\mathcal{I}$ in $poly(n, P)$ time. ∎

We now prove Theorem 5.1 in rest of the paper. We begin with some preliminaries which allow us to decompose the instance in a useful way.

Segments. A *segment* in the tree $\mathcal{T}$ of a `Demand MultiCut` instance is defined to be a maximal path for which all internal nodes have degree 2. Note there is a 1-1 correspondence between segments of $\mathcal{T}$ and edges of $\texttt{red}(\mathcal{T})$. See Figure 4 for an example.

Decomposing the instance. We begin by partitioning the paths in the input instance into two sets. The first set contains the paths which are confined to one segment, i.e., those paths P for which there is a segment S in $\mathcal{T}$ such that the edges of P are contained in S; the instance restricted to this set of paths will be called

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

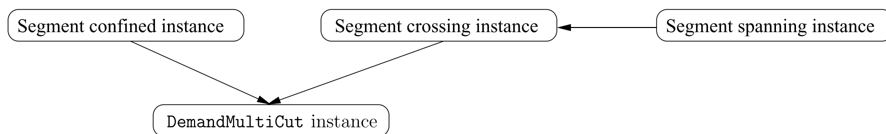

FIG. 5. *The roadmap for* `Demand MultiCut`.

the *segment confined* instance (see Figure 6). The second set contains the remaining paths; the instance restricted to this set of paths will be called the *segment crossing* instance.

We now consider the segment confined instance and the segment crossing instance separately which incurs a factor of 2 loss in the approximation ratio (see Figure 5 for a roadmap). We will use LP rounding techniques for the segment confined instance (subsection 5.1) and dynamic programming for the segment crossing instance (subsection 5.2).

5.1. Segment confined instances. In this section, we show that one can obtain constant factor polynomial time approximation algorithms for segment confined instances. In fact, this result follows from prior work on column restricted covering IPs [9]. 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 $e_1, \ldots, e_n$, and let the number of paths P be m. The following IP (IP3) captures the `Demand MultiCut` problem for $\mathcal{I}_S$:

$$\min \sum_{e \in S} c_e x_e, \tag{5.1}$$

$$\sum_{e \in P} p_e x_e \geq d(P) \qquad \text{for all paths } P \in \mathcal{P}_S, \tag{5.2}$$

$$x_e \in \{0, 1\} \quad \text{for all } e \in S. \tag{5.3}$$

Note that this is a covering integer program (CIP) 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 IPs (CIPs) as described in Chakrabarty, Grant, and Könemann [9], who developed the following framework for approximation algorithms for such an IP with the following guarantee. Consider the IP $\min c^T x$ such that $Ax \geq b, x \in \{0,1\}^n$ with $c, b \in \mathcal{Z}_+^n, A \in \mathcal{Z}_+^{m \times n}$ and with the special structure that for any column of A, the nonzero entries are all the same (denote this IP by IP*). Let us assume that we are the given following approximation guarantees for IPs derived from IP*:

1. **Assumption for 0-1 CIP.** Let A' be the matrix obtained by replacing all the nonzero entries of A by 1. Then, a 0-1 CIP is any IP of the form $\min w^T x$ such that $A'x \geq d, x \in \{0,1\}^n$ for any nonnegative vectors $w.d \in \mathcal{Z}_+^n$. We are given a polynomial time algorithm that finds a solution to any 0-1 CIP whose cost is within α-factor of the optimum of the canonical LP relaxation of the 0-1 CIP.
2. **Assumption for priority CIP.** Let $s \in \mathcal{Z}_+^n, \pi \in \mathcal{Z}_+^m$ which will be called priority supply and priority demand vectors, respectively. Construct the ma-

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

trix $A'[s,\pi]$ as $A'[s,\pi]_{i,j} = 1$ if $A_{i,j} = 1$ and $s_j \geq \pi_i$, and $A'[s,\pi]_{i,j} = 0$ otherwise. Then, a priority CIP is any IP of the form $\min w^T x$ such that $A'[s,d]x \geq 1, x \in \{0,1\}^n$ for any nonnegative vector $w \in \mathcal{Z}_+^n$. We are given a polynomial time algorithm that finds a solution to any priority CIP whose cost is within β-factor of the optimum of the canonical LP relaxation of the priority CIP.

With these assumptions about IP*, [9] shows the following result.

LEMMA 5.2 (see [9]). *There exists a polynomial time $24\alpha + 8\beta$-approximation algorithm for IP*.*

Applying this result for IP3, we will get a constant factor approximation for IP3 provided we can find polynomial time constant factor approximations for the following two problems (with the approximation guarantees being relative to the canonical LP relaxations of the respective problems): (i) 0-1 instances of `Demand MultiCut`, where the p_e values are either 0 or 1, (ii) priority version of `Demand MultiCut`, 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 the consecutive ones property. To see this, order the columns by the top-down ordering of the corresponding edges. Now, for any path, the edges contributing to it appear consecutively. Therefore, the LP relaxation has integrality gap of 1 since the constraint matrix is totally unimodular (and its optimum can be found in polynomial time).

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 consider the following LP (LP1):

$$\min \sum_{e \in S} c_e x_e, \tag{5.4}$$

$$\sum_{e \in P : p_e \geq p(P)} x_e \geq 1 \qquad \text{for all paths } P \in \mathcal{P}_S, \tag{5.5}$$

$$x_e \geq 0 \quad \text{for all } e \in S. \tag{5.6}$$

Note that this is (the relaxation of) a set cover problem. In general, a set cover LP might have $\Omega(\log n)$ integrality gap, but we will use the special structure of our instance to argue a $O(1)$ integrality gap. To this end, we will use the rounding framework for Chan et al. [10] who developed the parameter of *shallow cell complexity*, $f(t,k)$ (Definition 5.3) to capture the complexity of set cover instances.

DEFINITION 5.3 (shallow cell complexity). *Consider a set cover instance with element-set incidence matrix $A^{m\times n}$, i.e., the rows of A correspond to sets and the columns correspond to the elements and $A(i,j) = 1$ if set i contains element j. The shallow cell complexity of A is a function $f(t,k)$ (with $1 \leq t \leq m, 1 \leq k \leq n$) obtained as follows. $f(t,k)$ is the maximum over all submatrices A^* of A with exactly t columns, of the number of distinct rows of A^* with k or fewer ones (two rows of A^* are distinct if they are not the same as row vectors).*

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

[10] showed that for set cover instances of low shallow cell complexity, the canonical LP has small integrality gap. This is stated more formally in the following lemma.

LEMMA 5.4. *For set cover instances with shallow cell complexity $f(t,k) \leq t\phi(t)k^g$, there exists a polynomial time algorithm that rounds an optimum basic solution to the canonical LP for the set cover instance to within a factor $O(\max(1, \log \phi(t)))$.*

We now bound the shallow cell complexity of our instance.

CLAIM 5.5. *Let A be the constraint matrix of* LP1. *Then, the shallow cell complexity $f(t,k)$ of A is at most tk^2.*

Proof. Let A^* be a submatrix of A with exactly t columns. Columns of A correspond to edges in S. Contract all edges which do not contribute to columns of $A^\star$. Let $S^\star$ be the remaining (i.e., uncontracted) edges in S. Each path in $\mathcal{P}_S$ now maps to a new path obtained by contracting these edges. Let $\mathcal{P}^\star$ denote the set of resulting paths. For a path $P \in \mathcal{P}^\star$, let $E(P)$ be the edges in P whose priority is at least that of P. In the constraint matrix $A^\star$, the constraint for the path P has 1's in exactly the edges in $E(P)$. We can assume that the set $E(P)$ is distinct for every path $P \in \mathcal{P}^\star$ (because we are interested in counting the number of paths with distinct sets $E(P)$).

Let $\mathcal{P}^\star(k)$ be the paths in $\mathcal{P}^\star$ for which $|E(P)| \leq k$. We need to count the cardinality of this set. Fix an edge $e \in S^\star$; let $S^\star(e)$ be the edges in $S^\star$ of priority at least that of e. Let P be a path in $\mathcal{P}^\star(k)$ which has e as the least priority edge in $E(P)$ (breaking ties arbitrarily). Let e_l and e_r be the leftmost and the rightmost edges in $E(P)$, respectively. Note that $E(P)$ is exactly the edges in $S^\star(e)$ which lie between e_l and e_r. Since there are at most k choices for e_l and e_r (look at the k edges to the left and to the right of e in the set $S^\star(e)$), it follows that there are at most k^2 paths P in $\mathcal{P}^\star(k)$ which have e as the least priority edge in $E(P)$. For every path in $\mathcal{P}^\star(k)$, there are at most $|E^\star| = t$ choices for the least priority edge. Therefore the size of $\mathcal{P}^\star(k)$ is at most tk^2. ☐

It now follows from Lemma 5.4 that there is a polynomial time algorithm that rounds an optimum solution to LP1 to within a constant factor. Thus we obtain a constant factor approximation algorithm for segment confined instances.

5.2. Segment crossing instances on binary trees. In this section, we will obtain a $O(1)$ approximation for segment crossing instances.

To help develop some intuition, we first consider the special case in 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 $\texttt{red}(\mathcal{T})$. We call such an instance a *segment spanning instance* (see Figure 5 for the roadmap). An example is shown in Figure 6. Although we will not use results for this special case in the algorithm for the general case, many of the ideas will get extended to the general case.

5.2.1. Segment spanning instances. We will use dynamic programming (DP). For a vertex $v \in \texttt{red}(\mathcal{T})$, let $\mathcal{T}_v$ be the subtree 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 DP algorithm, we give some intuition for the DP table. We will consider subproblems which correspond to covering paths in $\mathcal{P}_v$ by edges in $\mathcal{T}_v$ for every vertex $v \in \texttt{red}(\mathcal{T})$. However, to solve this subproblem, we will also need to store the edges in $\mathcal{T}$ which are ancestors of v and are selected by our algorithm. Storing all

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

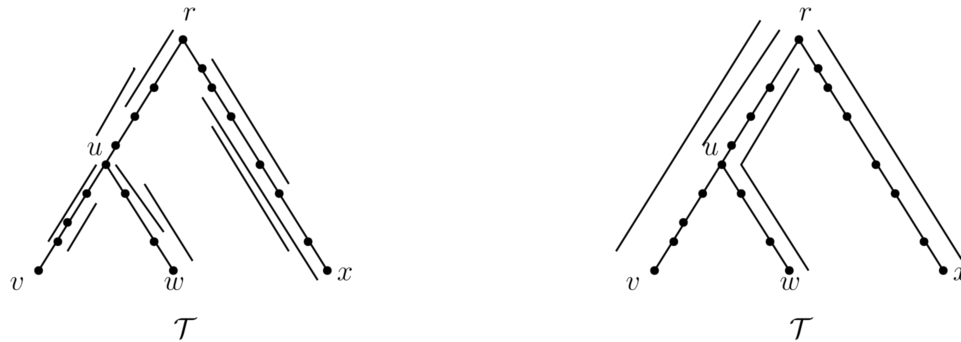

FIG. 6. *The left instance represents a segment confined instance, whereas the right one is a segment spanning instance. The lines represent the paths.*

such subsets would lead to too many DP table entries. Instead, we will work with the following idea—for each segment S, let $B^{\texttt{opt}}(S)$ be the total cost of edges in S which get selected by an optimal algorithm. If we know $B^{\texttt{opt}}(S)$, 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 $B^{\texttt{opt}}(S)$ (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 $B^{\texttt{opt}}(S)$ 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 $B^{\texttt{opt}}(S)$ in order of increasing density. Keep selecting them in this order till we exceed the budget $B^{\texttt{opt}}(S)$. Note that we pay at most twice of $B^{\texttt{opt}}(S)$ because the last edge will have cost at most $B^{\texttt{opt}}(S)$. 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 $S_1, \dots, S_k$ denote the segments which lie above v (in the order from the root to v), it will suffice if we store $B^{\texttt{opt}}(S_1), \dots, B^{\texttt{opt}}(S_k)$ with the DP table entry for v. We can further cut down the search space by assuming that each of the quantities $B^{\texttt{opt}}(S)$ 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 $B^{\texttt{opt}}(S_1), \dots, B^{\texttt{opt}}(S_k)$ is $O(\log^k n)$, because each of the quantities $B^{\texttt{opt}}(S_i)$ 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 $B^{\texttt{opt}}(S_i)$ and $B^{\texttt{opt}}(S_{i+1})$ 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.

DP algorithm. We first describe the greedy algorithm outlined above. The algorithm `GreedySelect` is given in Figure 7.

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

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

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

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.

FIG. 7. *Algorithm* `GreedySelect` *for selecting edges in a segment S with a budget B.*

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 $v \in \texttt{red}(\mathcal{T})$ at reduced depth k and a child w of v in $\texttt{red}(\mathcal{T})$ (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 $\texttt{red}(\mathcal{T})$ 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 8. We first compute the set $G(v)$ as outlined above. Let the children of v in the tree $\texttt{red}(\mathcal{T})$ be w_1 and w_2. Let the segments corresponding to (v, w_1) and (v, w_2) be 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 look up 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 will not analyze this algorithm here because its analysis will follow from the analysis of the more general case. 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 $2^{O(H)}$.

5.2.2. Arbitrary segment crossing instances. We will proceed as in the previous subsection, 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; 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

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Fill DP Table :

Input: A node $v \in \texttt{red}(\mathcal{T})$ at reduced depth k and
a state sequence $\Lambda_v = (B_1, \ldots, B_k)$.

0. If v is a leaf node, set $D[v, \Lambda_v]$ to 0, and exit.
1. Let $S_1, \ldots, S_k$ be the segments visited while going from 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 `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 ∞.
6. For $r = 1, 2$ (go to each of the two children and solve the subproblems)
   (i) For each extension $\Gamma_{w_r} = (B_1, \ldots, B_k, B^r_{k+1})$ of Γ_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. $D[v, \Lambda_v] \leftarrow M_1 + M_2$.

FIG. 8. *Filling a table entry $D[v, \Lambda_v]$ in the dynamic program.*

Algorithm `GreedySelect`**:**

Input: A cell (S, τ) and a budget B.

1. Initialize a set G to emptyset.
2. Let $S(\tau)$ be the edges in S of density class τ 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.

FIG. 9. *Algorithm* `GreedySelect` *for selecting edges in a segment S of density class τ with a budget B.*

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 its 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 9

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

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

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 that 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 to $3B$ 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 $S_1, \ldots, S_k$ 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), $i = 1, \ldots, k$, $\tau \in [\tau_{\min}, \tau_{\max}]$. 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 idea, we define the notion of a *valid cell sequence.*

DEFINITION 5.6 (valid cell sequence). *We say that a sequence $\mathcal{C}_v = \sigma_1, \ldots, \sigma_\ell$ 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 10. 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 goes either 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 $(S_i, \tau_1), (S_i, \tau_1 + 1), \ldots, (S_i, \tau_2)$ for some $\tau_1 \le \tau_2$. 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 10, 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.

DEFINITION 5.7 (valid segment budget sequence). *A valid segment budget sequence is a sequence $\Lambda_v^{\texttt{seg}} := (B_1^{\texttt{seg}}, \ldots, B_k^{\texttt{seg}})$, where $B_i^{\texttt{seg}}$ corresponds to the segment S_i. The sequence should satisfy the property that each of these quantities is a power of 2 and lies in the range $[1, 2n]$. Further, for any i, the ratio $B_i^{\texttt{seg}}/B_{i+1}^{\texttt{seg}}$ lies in the range $[1/8, 8]$.*

DEFINITION 5.8 (valid cell budget sequence). *Consider a valid cell sequence $\mathcal{C}_v = \sigma_1, \ldots, \sigma_\ell$. Then, a valid cell budget sequence is a sequence $\Lambda_v^{\texttt{cell}} := (B_1^{\texttt{cell}}, \ldots, B_\ell^{\texttt{cell}})$, where $B_j^{\texttt{cell}}$ corresponds to the cell σ_j. The sequence should satisfy the property that each of the quantities $B_j^{\texttt{cell}}$ lies in the range $[1, 2n]$. Further, for any j, the ratio $B_j^{\texttt{cell}}/B_{j+1}^{\texttt{cell}}$ 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. We are now ready to define the states for the DP table; we shall call these *valid states.*

DEFINITION 5.9 (valid state). *A* valid state $\texttt{State}(v)$ *at the vertex v is given by a triplet $(\mathcal{C}_v, \Lambda_v^{\texttt{seg}}, \Lambda_v^{\texttt{cell}})$ where $\mathcal{C}_v, \Lambda_v^{\texttt{seg}}, \Lambda_v^{\texttt{cell}}$ are a valid cell sequence, valid segment*

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

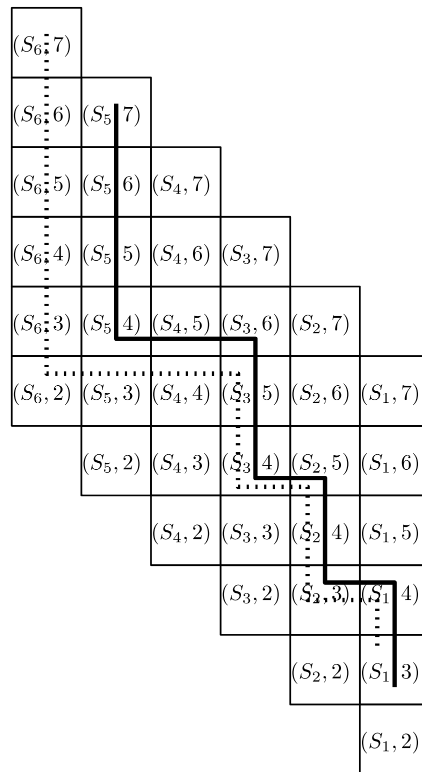

FIG. 10. *Let w be a vertex at reduced depth 6 and v be the parent of w in $\texttt{red}(\mathcal{T})$. The segments above w are labelled $S_1,\ldots,S_6$ (starting from the root downwards). The cells are arranged in a tabular fashion as shown—the density classes lie in the range $\{2,3,\ldots,7\}$. 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).*

budget sequence, and valid cell budget sequence, respectively. The triplet should further satisfy the property that for any cell $\sigma_j = (S_i,\tau)$ in $\mathcal{C}_v$, it must hold that $B_j^{\texttt{cell}} \le B_i^{\texttt{seg}}$.[2]

The intuition for the condition $B_j^{\texttt{cell}} \le B_i^{\texttt{seg}}$ in Definition 5.9 is that $B_j^{\texttt{cell}}$ corresponds to cost of density class τ edges in S_i, whereas $B_i^{\texttt{seg}}$ corresponds to the cost of all the edges in S_i (which are selected by the optimal solution).

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 the 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 subproblem 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 nontrivial fact is to show that maintaining

[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.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Algorithm `SelectSegment`**:**

Input: A vertex $v \in \texttt{red}(\mathcal{T})$, $\texttt{State}(v) := (\mathcal{C}_v, \Lambda_v^{\texttt{Seg}}, \Lambda_v^{\texttt{cell}})$, 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 $\texttt{GreedySelect}((S_i, \tau), B_j^{\texttt{cell}})$, where j is the index of (S_i, τ) in $\mathcal{C}_v$.
4. Add to G the edges returned by $\texttt{GreedySelect}((S_i, \tau_1), B_i^{\texttt{seg}})$.
5. Add to G all edges $e \in S_i$ of density class strictly less than τ and for which $c_e \leq B_i^{\texttt{seg}}$.
6. Return G.

FIG. 11. *Algorithm* `SelectSegment` *for selecting edges in a segment S as dictated by the state at v. The notations $B^{\texttt{seg}}$ and $B^{\texttt{cell}}$ are as explained in the text.*

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 11. 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. The reason for this will become clear during the proof of Claim 5.20.

Before specifying the DP table, we need to show what it means for a state to be an *extension* of another state. This is captured in the following definition.

DEFINITION 5.10. *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 10), 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 8. The details are given in Figure 12. This completes the description of the algorithm.

5.3. Algorithm analysis. We now analyze the algorithm.

Running time. We bound the running time of the algorithm. First we bound the number of possible table entries.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Fill DP Table :

Input: A node $v \in \texttt{red}(\mathcal{T})$ at reduced depth k, $\texttt{State}(v) = (\mathcal{C}_v, \Lambda_v^{\texttt{Seg}}, \Lambda_v^{\texttt{cell}})$.

0. If v is a leaf node, set $D[v, \texttt{State}(v)]$ 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 $\texttt{SelectSegment}(v, S_i, \texttt{State}(v))$.
   (ii) $G(v) \leftarrow G(v) \cup G_i(v)$.
4. Let w_1, w_2 be the two children of v in $\texttt{red}(\mathcal{T})$ and the corresponding segments be S_{k+1}^1, S_{k+1}^2.
5. Initialize M_1, M_2 to ∞.
6. For $r = 1, 2$ (go to each of the two children and solve the subproblems)
   (i) For each extension $\texttt{State}(w_r)$ of $\texttt{State}(v)$ do
      (a) Let $G_{k+1}(w_r)$ be the edges returned by $\texttt{SelectSegment}(w_r, S_{k+1}^r, \texttt{State}(w_r))$.
      (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) $M_r \leftarrow \min(M_r, \text{cost of } G_{k+1}(w_r) + D[w_r, \texttt{State}(w_r)])$.
7. $D[v, \Lambda_v] \leftarrow M_1 + M_2$.

FIG. 12. *Filling a table entry $D[v, \texttt{State}(v)]$ in the dynamic program.*

LEMMA 5.11. *For any vertex v, the number of possible valid states is $O((\log n)^2 \cdot 2^{O(H)} \cdot (\rho_{max}/\rho_{min})^2)$.*

Proof. The length of a valid cell sequence is bounded by $(\tau_{\max} - \tau_{min}) + 2H$. To see this, fix a vertex v at reduced depth k, with segments $S_1, \ldots, S_k$ from the root to the vertex v. Consider a valid cell sequence $\sigma_1, \ldots, \sigma_\ell$. For a cell $\sigma_j = (S_i, \tau)$, define a potential $\Phi_j = j + 2i + \tau$. We claim that the potential $\Phi_j = \Phi_{j+1}$ for all indices j in this sequence. To see this, there are two options for $\sigma_{j+1} = (S_{i'}, \tau')$:

- $S_i = S_{i'}, \tau' = \tau - 1$: Here, $\Phi_{j+1} = j + 1 + 2i + \tau - 1 = j + 2i + \tau = \Phi_j$.
- $S_{i'} = S_{i-1}, \tau' = \tau + 1$: Here $\Phi_{j+1} = j + 1 + 2(i-1) + \tau + 1 = \Phi_j$.

Therefore,

$$\ell + \tau_{\min} \leq \Phi_l = \Phi_0 \leq 2H + \tau_{\max}.$$

It follows that $\ell \leq 2H + \tau_{\max} - \tau_{\min}$. Given the cell σ_j, there are only two choices for σ_{j+1}. So, the number of possible valid cell sequences is bounded by $2^{2H+\tau_{\max}-\tau_{\min}} \leq 2^{2H} \cdot \rho_{\max}/\rho_{\min}$.

Now we bound the number of valid segment budget sequences. Consider such a sequence $B_1^{\texttt{seg}}, \ldots, B_k^{\texttt{seg}}$. Since $B_1^{\texttt{seg}} \in [1, 2n]$ and it is a power of 2, there are $O(\log n)$ choices for it. Given $B_i^{\texttt{seg}}$, there are at most 7 choices for $B_{i+1}^{\texttt{seg}}$, because $B_{i+1}^{\texttt{seg}}/B_i^{\texttt{seg}}$ is a power of 2 and lies in the range $[1/8, 8]$. Therefore, the number of such sequences is at most $O(\log n) \cdot 7^k \leq O(7^H \log n)$. Similarly, the number of valid cell budget sequences is at most $O(\log n) \cdot 7^\ell$, where ℓ is the maximum length of a valid cell sequence. By the argument above, ℓ is at most $2H + \tau_{\max} - \tau_{\min}$. Combining everything, we see that the number of possible states for v is bounded by a constant times

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

$$2^{2H} \cdot \rho_{\max}/\rho_{\min} \cdot 7^H \log n \cdot \log n \cdot 7^{2H} \cdot \rho_{\max}/\rho_{\min},$$

which implies the desired result. ∎

We can now bound the running time easily.

LEMMA 5.12. *The running time of the algorithm is polynomial in $n, 2^H, \rho_{\max}/\rho_{\min}$.*

Proof. To fill the table entry for $D[v, \Gamma(v)]$, where v has children w_1, w_2, the algorithm in Figure 12 cycles through the number of possible extensions of $\Gamma(v)$ for each of the two children. Since any valid extension of $\Gamma(v)$ for a child w_r is also a valid state at w_r, the result follows from Lemma 5.11. ∎

Feasibility. We now argue that the table entries in the DP correspond to valid solutions. Fix a vertex $v \in \mathtt{red}(\mathcal{T})$, and let $S_1, \dots, S_k$ be the segments as we go from the root to v. Recall that $\mathcal{T}(v)$ denotes the subtree of $\mathcal{T}$ rooted below v and $\mathcal{P}(v)$ denotes the paths in $\mathcal{P}$ which have an internal vertex in $\mathcal{T}(v)$. For a segment S and density class τ, let $S(\tau)$ denote the edges of density class τ in S.

LEMMA 5.13. *Consider the algorithm in Figure 12 for filling the DP entry $D[v, \mathtt{State}(v)]$, and let $G(v)$ be the set of vertices obtained after step 3 of the algorithm. Assuming that this table entry is not ∞, there is a subset $Y(v)$ of edges in $\mathcal{T}(v)$ such that the cost of $Y(v)$ is at equal to $D[v, \mathtt{State}(v)]$ and $Y(v) \cup G(v)$ is a feasible solution for the paths in $\mathcal{P}(v)$.*

Proof. We prove this by induction on the reduced height of v. If v is a leaf, then $\mathcal{P}(v)$ is empty, and so the result follows trivially. Suppose it is true for all nodes in $\mathtt{red}(\mathcal{T})$ at reduced height at most $k-1$, and let v be at height k in $\mathtt{red}(\mathcal{T})$. We use the notation in Figure 12. Consider a child w_r of v, where r is either 1 or 2. Let the value of M_r used in step 7 be equal to the cost of $G_{k+1}(w_r) + D[w_r, \mathtt{State}(w_r)]$ for some $\mathtt{State}(w_r)$ given by $(\mathcal{C}_{w_r}, \Lambda^{\mathtt{seg}}_{w_r}, \Lambda^{\mathtt{cell}}_{w_r})$ with $\Lambda^{\mathtt{seg}}_v = (B^{\mathtt{seg}}_1, \dots, B^{\mathtt{seg}}_k)$, $\Lambda^{\mathtt{seg}}_{w_r} = (B^{\mathtt{seg}'}_1, \dots, B^{\mathtt{seg}'}_{k+1})$, and $\Lambda^{\mathtt{cell}}_{w_r} = (B^{\mathtt{cell}'}_1, \dots, B^{\mathtt{cell}'}_\ell)$. Let $G(v)$ and $G_{k+1}(w_r)$ be as in the steps 3 and 6(i)(a), respectively. We ensure that $G(v) \cup G_{k+1}(w_r)$ covers all paths in $\mathcal{P}(v)$ which end before w_r. The following claim is the key to the correctness of the algorithm.

CLAIM 5.14. *Let $G(w_r)$ be edges obtained at the end of step 3 in the algorithm in Figure 12 when filling the DP table entry $D[w_r, \mathtt{State}(w_r)]$. Then $G(w_r)$ is a subset of $G(v) \cup G_{k+1}(w_r)$.*

Proof. Let S^r_{k+1} be the segment between v and w_r. By definition, $G_{k+1}(w_r)$ and $G(w_r) \cap S^r_{k+1}$ are identical. Let us now worry about segments $S_i, i \le k$. Fix such a segment S_i.

We know that after the cells corresponding to the segment S^r_{k+1}, the sequence $\mathcal{C}_{w_r}$ lies below $\mathcal{C}_v$ till it meets $\mathcal{C}_v$. Now consider various case for an arbitrary cell $\sigma = (S_i, \tau)$ (we refer to the algorithm `SelectSegment` in Figure 11):

- The cell σ lies above $\mathcal{C}_{w_r}$: $G(w_r)$ does not contain any edge of $S_i(\tau)$.
- The cell σ lies below $\mathcal{C}_{w_r}$: The cell will lie below $\mathcal{C}_v$ as well, and so $G(v)$ and $G(w)$ will contain the same edges from $S_i(\tau)$ (because $B^{\mathtt{seg}}_i = B^{\mathtt{seg}'}_i$ are the same).
- The cell σ lies on $\mathcal{C}_{w_r}$: If it also lies on $\mathcal{C}_v$, then the fact that $B^{\mathtt{cell}}$ and $B^{\mathtt{cell}'}$ values for this cell are the same implies that $G(v)$ and $G(w_r)$ pick the same edges from $S_i(\tau)$ (in case τ happens to be the smallest indexed density class for which $(S_i, \tau) \in \mathcal{C}_{w_r}$, then the same will hold for $\mathcal{C}_v$ as well). If it lies below

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

$\mathcal{C}_v$, then the facts that $B_i^{\texttt{seg}} = B_i^{\texttt{seg}'}$ and $B_i^{\texttt{seg}} \geq B_j^{\texttt{cell}} = B_{j'}^{\texttt{cell}'}$, where j and j' are the indices of this cell in the two cell sequences, respectively, imply that $G(v)$ will pick all the edges fof cost at most $B_i^{\texttt{seg}}$ from $S_i(\tau)$, whereas $G(w_r)$ will pick only a subset of these edges. □

We see that $G(v) \cap S_i$ contains $G(w_r) \cap S_i$. This proves the claim.

By induction hypothesis, there is a subset $Y(w_r)$ of edges in the subtree $\mathcal{T}(w_r)$ of cost equal to $D[w_r, \texttt{State}(w_r)]$ such that $Y(w_r) \cup G(w_r)$ satisfies all paths in $\mathcal{P}(w_r)$. We already know that $G(v) \cup G_{k+1}(w_r)$ covers all paths in $\mathcal{P}(v)$ which end in the segment S_{k+1}^r. Since any path in $\mathcal{P}(v)$ will either end in S_{k+1}^1 or S_{k+1}^2 or will belong to $\mathcal{P}(w_1) \cup \mathcal{P}(w_2)$, it follows that all paths in $\mathcal{P}(v)$ are covered by $\cup_{r=1}^2 (Y(w_r) \cup G(w_r) \cup G_{k+1}(w_r)) \cup G(v)$. Now, the claim above shows that $G(w_r) \subseteq G_{k+1}(w_r) \cup G(v)$. So this set is the same as $Y(w_1) \cup Y(w_2) \cup G_{k+1}(w_1) \cup G_{k+1}(w_2) \cup G(v)$ (and these sets are mutually disjoint). Recall that M_r is equal to the cost of $G_{k+1}(w_r) \cup Y(w_r)$; it follows that the the DP table entry for v for these parameters is exactly the cost of $Y(w_1) \cup Y(w_2) \cup G_{k+1}(w_1) \cup G_{k+1}(w_2)$. This proves the lemma. □

For the root vertex r, a valid state at r must be the empty set. The above lemma specialized to the root r implies the following.

Corollary 5.15. *Assuming that $D[r, \emptyset]$ is not ∞, it is the cost of a feasible solution to the input instance.*

Approximation ratio. Now we relate the values of the DP table entries to the values of the optimal solution for suitable subproblems. We give some notation first. Let $\texttt{OPT}$ denote an optimal solution to the input instance. For a segment S, we shall use $S^{\texttt{opt}}$ to the denote the subset of S selected by $\texttt{OPT}$. Similarly, let $S^{\texttt{opt}}(\tau)$ denote the subset of $S(\tau)$ selected by $\texttt{OPT}$. Let $B^{\texttt{opt}}(S,\tau)$ denote the total cost of edges in $S^{\texttt{opt}}(\tau)$ and $B^{\texttt{opt}}(S)$ denote the cost of edges in $S^{\texttt{opt}}$.

We first describe a high-level plan for the argument. We will begin by "smoothing" the optimal solution, i.e., replacing the values $B^{\texttt{opt}}(S,\tau)$ at $O(1)$ loss by values $B^\star(S,\tau)$ so that the values $B^\star(S,\tau)$ don't vary too rapidly across neighboring values of τ or neighboring segments. We will then compare our solution costs to the smoothed costs $B^\star$ in the following manner: for each vertex $v \in \texttt{red}(\mathcal{T})$, we will use the values $B^\star(S,\tau)$ to construct a special state $\texttt{State}^\star(v)$. The special states $\texttt{State}^\star(v)$ will have the property that when our algorithm in Figure 12 fills the table entry $D[v, \texttt{State}^\star(v)]$, it will incur cost at most a constant times the total of the $B^\star(S,\tau)$ values for all the cells (S,τ) in the subtree rooted at v.

We first show that we can upper bound $B^{\texttt{opt}}(S,\tau)$ by values $B^\star(S,\tau)$ values such that the latter values are close to each other for nearby cells. For two segments S and S', define the distance between them as the distance between the corresponding edges in $\texttt{red}(T)$ (the distance between two adjacent edges is 1). Similarly, we say that a segment is the parent of another segment if this relation holds for the corresponding edges in $\texttt{red}(\mathcal{T})$. Let $\texttt{cells}(\mathcal{T})$ denote the set of all cells in $\mathcal{T}$.

Lemma 5.16. *We can find values $B^\star(S,\tau)$ for each cell (S,τ) such that the following properties are satisfied:*

i. *for every cell (S,τ), $B^\star(S,\tau)$ is a power of 2, and $B^\star(S,\tau) \geq B^{\texttt{opt}}(S,\tau)$;*

ii.

$$\sum_{(S,\tau)\in\texttt{cells}(\mathcal{T})} B^\star(S,\tau) \leq 16 \cdot \sum_{(S,\tau)\in\texttt{cells}(\mathcal{T})} B^{\texttt{opt}}(S,\tau);$$

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

iii. (smoothness) *for every pair of segments S, S', where S' is the parent of S, and density class τ, $8B^\star(S,\tau+1) \geq B^\star(S,\tau) \geq B^\star(S,\tau+1)/8$ and $8B^\star(S',\tau) \geq B^\star(S,\tau) \geq B^\star(S',\tau)/8$.*

Proof. We define

$$B^\star(S,\tau) := \sum_{i\geq 0} \sum_{S' \in N_i(S)} \sum_j \frac{B^{\texttt{opt}}(S', \tau+i+j)}{4^{i+|j|}},$$

where i varies over nonnegative integers, j varies over integers, and the range of i, j are such that $\tau + i + j$ remains a valid density class; and $N_i(S)$ denotes the segments which are at distance at most i from S. Note that $B^\star(S,\tau)$ is a not a power of 2 yet, but we will round it up later. As of now, $B^\star(S,\tau) \geq B^{\texttt{opt}}(S,\tau)$ because the term on RHS for $i = 0, j = 0$, is exactly $B^{\texttt{opt}}(S,\tau)$.

We now verify the second property. We add $B^\star(S,\tau)$ for all the cells (S,τ). Let us count the total contribution towards terms containing $B^{\texttt{opt}}(S',\tau')$ on the RHS. For every segment $S \in N_i(S')$ and density class $\tau' - i - j$, it will receive a contribution of $\frac{1}{4^{i+|j|}}$. Since $|N_i(s')| \leq 2^{i+1}$ (this is where we are using the fact that $\mathcal{T}$ is binary), this is at most

$$\sum_{i\geq 0}\sum_j \frac{2^{i+1}}{4^{i+|j|}} \leq \sum_{i\geq 0} \frac{2^{i+2}}{4^i} \leq 8.$$

Now consider the third condition. Consider the expressions for $B^\star(S,\tau)$ and $B^\star(S',\tau)$ where S' is the parent of S. If a segment is at distance i from S, its distance from S' is either i or $i \pm 1$. Therefore, the coefficients of $B^{\texttt{opt}}(S'',\tau'')$ in the expressions for $B^\star(S,\tau)$ and $B^\star(S',\tau)$ will differ by a factor of at most 4. The same observation holds for $B^\star(S,\tau)$ and $B^\star(S,\tau+1)$. It follows that

$$4B^\star(S,\tau+1) \geq B^\star(S,\tau) \geq B^\star(S,\tau+1)/4, \text{ and } 4B^\star(S',\tau) \geq B^\star(S,\tau) \geq B^\star(S',\tau)/4.$$

Finally, we round all the $B^\star(S,\tau)$ values up to the nearest power of 2. We will lose an extra factor of 2 in the statements (ii) and (iii) above. ∎

We will use the definition of $B^\star(S,\tau)$ in the lemma above for rest of the discussion. For a segment S, define $B^\star(S)$ as the maximum over all density classes τ of $B^\star(S,\tau)$. The following corollary follows immediately from the lemma above.

COROLLARY 5.17. *Let S and S' be two segments in $\mathcal{T}$ such that S' is the parent of S. Then $B^\star(S)$ and $B^\star(S')$ lie within a factor of 8 of each other.*

Proof. Let $B^\star(S)$ be equal to $B^\star(S,\tau)$ for some density class τ. Then,

$$B^\star(S) = B^\star(S,\tau) \overset{Lemma\ 5.16}{\leq} 8B^\star(S',\tau) \leq 8B^\star(S').$$

The other part of the argument follows similarly. ∎

The plan now is to define a valid state $\texttt{State}^\star(v) = (\mathcal{C}^\star_v, \Lambda^{\star\texttt{seg}}_v, \Lambda^{\star\texttt{cell}}_v)$ for each of the vertices v in $\texttt{red}(\mathcal{T})$. We begin by defining a *critical density* $\tau^\star(S)$ for each segment S. Recall that $S(\tau)$ denotes the edges of density class τ in S. Let $S(\leq \tau)$ denote the edges class of density class at most τ in S. For a density class τ and a budget B, let $S(\leq \tau, \leq B)$ denote the edges in $S(\leq \tau)$ which have cost at most B. Define $\tau^\star$ as the smallest density class τ such that the total cost of edges in $S(\leq \tau, \leq B^\star(S))$ is at least $4B^\star(S) + \sum_{\tau' \leq \tau} B^{\texttt{opt}}(S,\tau')$ (if no such density class exists, set τ to $\tau_{\max}$). Intuitively, we are trying to augment the optimal solution by low density edges, and

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

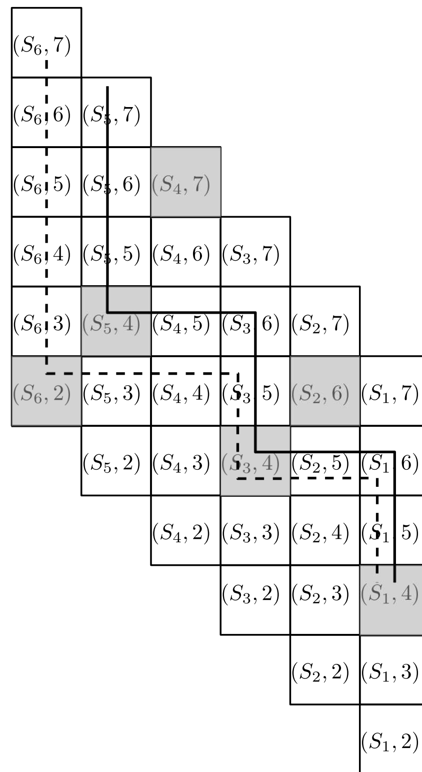

Fig. 13. *Refer to the notation used in Figure* 10. *The shaded cells represnt the critical density class for the corresponding segment. The solid line shows* $\mathcal{C}_v^\star$, *and the dotted line shows* $\mathcal{C}_w^\star$. *As an example, the cell* $(S_4, 4)$ *dominates the cells* $(S_3, 6), (S_3, 7)$, *and* $(S_2, 7)$.

$\tau^\star(S)$ tells us the density class till which we can essentially take all the edges in S (provided we do not pick any edge which is too expensive).

Having defined the notion of critical density, we are now ready to define a valid state $\texttt{State}^\star(v)$ for each vertex v in $\texttt{red}(\mathcal{T})$. Let v be such a vertex at reduced depth k, and let $S_1, \ldots, S_k$ be the segments starting from the root to v. Again, it is easier to see the definition of the cell sequence $\mathcal{C}_v^\star$ pictorially. As in Figure 13, the cell sequence starts with $(S_k, \tau_{\max})$ and keeps going down till it reaches the cell $(S_k, \tau^\star(S_k))$. Now it keeps going right as long as the cell corresponding to the critical density lies above it. If this cell lies below it, it moves down. The formal procedure for constructing this path is given in Figure 14. For sake of brevity, let $\tau_i^\star$ denote $\tau^\star(S_i)$.

We shall denote the sequence $\mathcal{C}_v^\star$ by $\sigma_1^\star, \ldots, \sigma_\ell^\star$. The corresponding segment budget sequence and cell budget sequences are easy to define.

Define $\Lambda_v^{\star\texttt{seg}} = (B_1^{\star\texttt{seg}}, \ldots, B_k^{\star\texttt{seg}})$, where $B_i^{\star\texttt{seg}} := B^\star(S_i)$. Similarly, define $\Lambda_v^{\star\texttt{cell}} = (B_1^{\star\texttt{cell}}, \ldots, B_\ell^{\star\texttt{cell}})$ such that for the cell $\sigma_j^\star = (S_i, \tau)$, $B_j^{\star\texttt{cell}} := B^\star(S_i, \tau)$. This completes the definition of $\texttt{State}^\star(v)$. It is easy to check that these are valid sequences. Indeed, Lemma 5.16 and Corollary 5.17 show that each of the quantities $B_i^{\star\texttt{seg}}, B_j^{\star\texttt{cell}}$ are at most $2n$, and two such consecutive quantities are within factor of 8 of each other.

Further, let w be a child of v in $\texttt{red}(\mathcal{T})$. It is again easy to see that $\texttt{State}^\star(w)$ is an extension of $\texttt{State}^\star(v)$. The procedure for constructing $\mathcal{C}_w^\star$ ensures that this property holds: this path first goes down till $(S_{k+1}, \tau^\star(S_{k+1}))$, where S_{k+1} is the segment between v and w. Subsequently, it moves right till it hits $\mathcal{C}_v^\star$ (see Figure 13

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Construct Sequence $\mathcal{C}_v^\star$:

Input: A node $v \in \texttt{red}(\mathcal{T})$ at depth k, integers $\tau_1^\star, \ldots, \tau_k^\star$

1. Initialize $\mathcal{C}_v^\star$ to empty sequence, and $i \leftarrow k, \tau \leftarrow \tau_{\max}$
2. While $(i \geq 1)$
   (i) Add the cell (S_i, τ) to $\mathcal{C}_v^\star$.
   (ii) If $\tau > \tau_i^\star$, $\tau \leftarrow \tau - 1$
   (iii) Else $i = i - 1, \tau \leftarrow \tau + 1$.

Fig. 14. *Construction of the path $\mathcal{C}_v^\star$.*

for an example). The following crucial lemma shows that it is alright to ignore the cells above the path $\mathcal{C}_v^\star$. Let w_1 and w_2 be the children of v in $\texttt{red}(\mathcal{T})$. We consider the algorithm in Figure 12 for filling the DP entry $D[v, \texttt{State}^\star(v)]$. Let $G^\star(v)$ be the edges obtained at the end of step 3 in this algorithm. Further, let $G_{k+1}^\star(w_r)$ be the set of edges obtained in step 6(i)(a) of this algorithm when we use the extension $\texttt{State}^\star(w_r)$.

Lemma 5.18. *For $r = 1, 2$, any path in $\mathcal{P}(v)$ which ends in the segment S_{k+1}^r is satisfied by $G^\star(v) \cup G_{k+1}^\star(w_r)$.*

This is the main technical lemma of the contribution and is the key reason why the algorithm works. We will show this by a sequence of steps. We say that a cell (S_i, τ) *dominates* a cell (S_j, τ') if $j < i$ and $\tau' - \tau > j - i$. As in Figure 13, a cell (S, τ) dominates all cells which lie in the upper right quadrant with respect to it if we arrange the cells as shown in the figure. For a segment S_i, let $\texttt{Dom}(S_i)$ be the set of cells dominated by $(S_i, \tau_i^\star)$. The following claim shows why this notion is useful. For a set E of edges , let $p(E)$ denote $\sum_{e \in E} p_e$. Recall that $S^{\texttt{opt}}(\tau)$ denotes the set of edges in $S(\tau)$ selected by the optimal solution.

Claim 5.19.

$$\sum_{(S_j, \tau) \in \texttt{Dom}(S_i)} p\left(S_j^{\texttt{opt}}(\tau)\right) \leq 2 \cdot 128^{-\tau_i^\star} B^\star(S_i, \tau_i^\star).$$

Proof. Fix a segment S_j. For sake of brevity, let $\bar{\tau}$ denote $\tau_i^\star + (i - j)$. Recall that for any pair (S, τ), $B^\star(S, \tau) \geq B^{\texttt{opt}}(S, \tau)$ (Lemma 5.16). Therefore, terms in the above sum corresponding to S_j add up to

$$\sum_{\tau \geq \bar{\tau}} p\left(S_j^{\texttt{opt}}(\tau)\right) \leq \sum_{\tau \geq \bar{\tau}} 128^{-\tau} B^\star(S_j, \tau).$$

By repeated applications of Lemma 5.16,

$$B^\star(S_j, \tau) \leq 8^{\tau - \tau_i^\star} \cdot 8^{i-j} B^\star(S_i, \tau_i^\star).$$

Therefore,

$$\begin{aligned}
\sum_{\tau \geq \bar{\tau}} p(S_j^{\texttt{opt}}(\tau)) &\leq \sum_{\tau \geq \bar{\tau}} 128^{-\tau} \cdot 8^{\tau - \tau_i^\star} \cdot 8^{i-j} B^\star(S_i, \tau_i^\star) \\
&= 128^{-\tau_i^\star} \sum_{\tau \geq \bar{\tau}} 16^{-(\tau - \tau_i^\star)} \cdot 8^{i-j} B^\star(S_i, \tau_i^\star) \\
&\leq 128^{-\tau_i^\star} \cdot \frac{2 \cdot 8^{i-j}}{16^{\bar{\tau} - \tau_i^\star}} \cdot B^\star(S_i, \tau_i^\star) = 128^{-\tau_i^\star} \cdot \frac{2}{2^{i-j}} \cdot B^\star(S_i, \tau_i^\star).
\end{aligned}$$

Summing over all $j < i$ now implies the result. ∎

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Let $G_i^\star(v)$ denote the set of edges selected by the algorithm SelectSegment in Figure 11 for the vertex v and segment S_i when called with the state $\texttt{State}^\star(v)$. Let $G_i^\star(\tau, v)$ be the density τ edges in $G_i^\star(v)$. The following claim shows that the total size of edges in it is much larger than the corresponding quantity for the optimal solution.

CLAIM 5.20. *For any segment S_i, when $\tau_i^\star < \tau_{\max}$,*

$$\sum_{\tau \le \tau_i^\star} p\left(G_i^\star(\tau, v)\right) - \sum_{\tau \le \tau_i^\star} p(S_i^{\texttt{opt}}(\tau)) \ge 2 \cdot 128^{-\tau_i^\star} B^\star(S_i, \tau_i^\star).$$

Proof. Recall that for a segment S, density class τ, and budget B, $S(\tau, \le B)$ denotes the edges in $S(\tau)$ which have cost at most B. The quantity $S(\le \tau, \le B)$ was defined similarly for edges of density class at most τ in S.

Consider the algorithm SelectSegment for S_i with the parameters mentioned above. Note that $\tau_i^\star$ is the same as τ_1 in the notation used in Figure 11. Clearly, for $\tau < \tau_i^\star$, the algorithm ensures that $G_i^\star(\tau, v)$ contains $S_i^{\texttt{opt}}(\tau)$ (because it selects all edges in $S_i(\tau, \le B^\star(S_i))$. Since each edge in $S_i^{\texttt{opt}}(\tau)$ has cost at most $B^{\texttt{opt}}(S_i, \tau) \le B^\star(S_i)$, this implies that $S_i(\tau, \le B^\star(S_i))$ contains $S_i^{\texttt{opt}}(\tau))$. For the class τ_1, note that the algorithm tries to select edges of total cost at least $4B^\star(S_i)$. Two cases arise: (i) if it is able to select these many edges, then the fact that the optimal solution selects edges of total cost at most $B^\star(S_i)$ from $S_i(\tau_1)$ implies the result; or (ii) the total cost of edges in $S_i(\tau_1, \le B^\star(S_i))$ is less than $4B^\star(S_i)$: in this case the algorithm selects all the edges from $S_i(\le \tau_1, \le B^\star(S_i))$, and so $G_i^\star(\tau, v)$ contains $S_i^{\texttt{opt}}(\tau)$ for all $\tau \le \tau_i^\star$. The definition of $\tau_i^\star$ implies that the total cost of edges in $\cup_{\tau \le \tau_i^\star} G_i^\star(\tau, v) \setminus S_i^{\texttt{opt}}(\tau)$ is at least $4B^\star(S_i)$, and so the result follows again. ∎

We are now ready to prove Lemma 5.18.

Proof of Lemma 5.18. Let P be a path in $\mathcal{P}(v)$ which ends in the segment S_{k+1}^r. Suppose P starts in the segment S_{i_0}. Note that P contains the segments $S_{i_0+1}, \ldots, S_k$ but may partially intersect S_{i_0} and S_{k+1}.

CLAIM 5.21. *For a cell (S_i, τ) on the cell sequence $\mathcal{C}^\star(v)$, $p(G_i^\star(\tau, v) \cap P) \ge p(S_i^{\texttt{opt}}(\tau) \cap P)$.*

Proof. Fix a segment S_i which is intersected by P. P contains the lower end-point of this segment S_i. If $S_i^{\texttt{opt}}(\tau) \subseteq G_i^\star(\tau, v)$, there is nothing to prove. Else let e be the first edge in $S_i^{\texttt{opt}}(\tau) \setminus G_i^\star(\tau, v)$ as we go up from the lower end-point of this segment. It follows that during step 5 of the algorithm GreedySelect in Figure 9, we would select edges of total cost at least $2B^{\texttt{opt}}(S_i, \tau)$ (because $B^{\texttt{opt}}(S_i, \tau) \le B^\star(S_i, \tau)$). The claim follows. ∎

Clearly, if (S_i, τ) lies below the cell sequence $\mathcal{C}^\star(v)$, $G_i^\star(\tau, v)$ contains $S_i^{\texttt{opt}}(\tau)$ (because $G_i^\star(\tau, v)$ is the same as $S_i(\tau, \le B^\star(S_i))$, and $B^\star(S_i) \ge B^{\texttt{opt}}(S_i)$). Let $\texttt{Above}(\mathcal{C}^\star(v))$ denotes the cells lying above this sequence and $\texttt{Below}(\mathcal{C}^\star(v))$ the ones lying below it. The above claim now implies that

$$\sum_{\tau:(S_{i_0}, \tau) \in \mathcal{C}^\star(v) \cup \texttt{Below}(\mathcal{C}^\star(v))} p\left(S_{i_0}^{\texttt{opt}}(\tau) \cap P\right) \le \sum_{\tau:(S_{i_0}, \tau) \in \mathcal{C}^\star(v) \cup \texttt{Below}(\mathcal{C}^\star(v))} p\left(G_{i_0}^\star(\tau, v) \cap P\right). \tag{5.7}$$

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Note that any cell (S_i, τ), $i \geq i_0$, lying above $\mathcal{C}^\star(v)$ must be dominated by one of the cells $(S_{i'}, \tau^\star_{i'})$ for $i' = i_0 + 1, \ldots, k$ with $\tau^\star_{i'} < \tau_m ax$. Therefore, Claim 5.19 shows that

$$\sum_{(S_i,\tau)\in \mathtt{Above}(\mathcal{C}^\star(v)), i_0 \leq i \leq k} p\left(S_i^{\mathtt{opt}}(\tau)\right) \leq \sum_{i: i_0 < i \leq k \text{ and } \tau^\star_i < \tau_{\max}} 2 \cdot 128^{-\tau^\star_i} B^\star(S_i, \tau^\star_i). \tag{5.8}$$

Further, for cells lying on or below $\mathcal{C}^\star(v)$, we get using Claim 5.20 and Claim 5.21 that

$$\sum_{(S_i,\tau)\in \mathcal{C}^\star(v)\cup\mathtt{Below}(\mathcal{C}^\star(v)), i_0 < i \leq k} p\left(S_i^{\mathtt{opt}}(\tau)\right) \leq \sum_{(S_i,\tau)\in \mathcal{C}^\star(v)\cup\mathtt{Below}(\mathcal{C}^\star(v)), i_0 < i \leq k} G^\star_i(\tau, v) - \sum_{i: i_0 < i \leq k \text{ and } \tau^\star_i < \tau_{\max}} 2 \cdot 128^{-\tau^\star_i} B^\star(S_i, \tau^\star_i). \tag{5.9}$$

Adding the three inequalities above, we see that $\sum_{i=i_0}^k p(S_i^{\mathtt{opt}} \cap P)$ is at most $\sum_{i=i_0}^k p(G^\star_i(v))$. It remains to consider segment S^r_{k+1}. In an argument identical to the one in Claim 5.21, we can argue that $p(G^\star_{k+1}(w_r) \cap P) \geq p(S^{r,\mathtt{opt}}_{k+1} \cap P)$, where $S^{r,\mathtt{opt}}_{k+1}$ denotes the edges in S^r_{k+1} selected by the optimal solution. This completes the proof of the technical Lemma 5.18. ∎

The rest of the task is now easy. We just need to show that DP table entries corresponding to these valid states are comparable to the cost of the optimal solution. For a vertex $v \in \mathtt{red}(\mathcal{T})$, we shall use the notation $\mathtt{cells}(\mathcal{T}(v))$ to denote the cells (S, τ), where S lies in the subtree $\mathcal{T}(v)$.

LEMMA 5.22. *For every vertex v, the table entry $D[v, \mathtt{State}^\star(v)]$ is at most $20 \sum_{(S,\tau)\in\mathtt{cells}(\mathcal{T}(v))} B^\star(S, \tau)$.*

Proof. We prove by induction on the reduced depth of v. If v is a leaf, the lemma follows trivially. Now suppose v has children w_1 and w_2. Consider the iteration of step 6 in the algorithm in Figure 12, where we try the extension $\mathtt{State}^\star(w_r)$ of the child w_r. Lemma 5.18 shows that in step 6(b), we will satisfy all paths in $\mathcal{P}(v)$ which end in the segment S^r_{k+1}. We now bound the cost of edges in $G^r_{k+1}(w_r)$ defined in step 6(a).

CLAIM 5.23. *The cost of $G^r_{k+1}(w_r)$ is at most $20 \sum_\tau B^\star(S^r_{k+1}, \tau)$.*

Proof. We just need to analyze the steps in the algorithm $\mathtt{SelectSegment}$ in Figure 11. For sake of brevity, let $\tau^\star$ denote $\tau^\star(S^r_{k+1})$ and $B^\star$ denote $B^\star(S^r_{k+1})$. The definition of $\tau^\star$ shows that the total cost of edges in $S^r_{k+1}(\leq \tau^\star, \leq B^\star)$ is at most $\sum_{\tau \leq \tau^\star} B^{\mathtt{opt}}(S^r_{k+1}, \tau) + 4B^\star \leq \sum_{\tau\leq\tau^\star} B^\star(S^r_{k+1}, \tau) + 4B^\star$. For the density class $\tau^\star$, the set of edges selected would cost at most $6B^\star$, because the algorithm in Figure 9 will take edges of cost up to $3B^\star$ from either end. Similarly, for density classes τ more than $\tau^\star$, this quantity is at most $6B^\star(S, \tau)$. Summing up everything, and using the fact that $B^\star = B^\star(S, \tau)$ for some density class τ, gives the result. ∎

The lemma now follows by applying induction on $D[w_r, \mathtt{State}^\star(w_r)]$. ∎

Applying the above lemma to the root vertex r, we see that $D[r, \emptyset]$ is at most a constant times $\sum_{(S,\tau)} B^\star(S, \tau)$, which by Lemma 5.16 is a constant times the optimal cost. Finally, Corollary 5.15 shows that this entry denotes the cost of a feasible solution. Thus, we have shown the main Theorem 5.1.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

6. Extensions.

6.1. Extension to ℓ_p norm of weighted flow-times. We sketch in this section how our techniques naturally extend to an $O(1)$-approximation for the problem of minimizing the ℓ_p norm of weighted flow-times for any $p \geq 1$ (`pWtdFlowTime`).[3] More formally, we have the following theorem.

THEOREM 6.1. *There is a constant factor approximation algorithm for* `pWtdFlowTime` *where the running time of the algorithm is polynomial in n and P.*

Our idea will be, again, to reduce `pWtdFlowTime` to `Demand MultiCut` at $O(1)$ loss and then apply Theorem 5.1.

We now sketch the reduction which is analogous to the arguments in sections 2.1 and 4. We start with an IP which has the same constraints as the integer program in subsection 2.1 but with the objective function changed to $\sum_j (\sum_{t \in [r_j, T]} w_j x_{j,t})^p$. The new objective function corresponds to the pth power of the ℓ_p norm of weighted flow-times. For this IP (call it IP4) the following analogue of Theorem 2.1 can be shown to hold.

THEOREM 6.2. *Suppose $x_{j,t}$ is a feasible solution to* IP4*. Then, there is a schedule for which the pth power of the ℓ_p norm of weighted flow-times is equal to the cost of the solution $x_{j,t}$.*

Now we use the same ideas as in subsection 3 to write a new weaker IP (call it IP5) with more structure. IP5 has the same constraints as IP2 but with the definition of $w(j, S)$ changed to $(w_j \cdot l(S))^p$. We can now prove the following analogue of Lemma 3.2.

LEMMA 6.3. *Given a solution x for* IP4*, we can construct a solution for* IP5 *of cost at most $4 \cdot 2^p$ times the cost of x. Similarly, given a solution y for* IP5*, we can construct a solution for* IP4 *of cost at most 4^p times the cost of y.*

IP5 can now be viewed as modeling `Demand MultiCut` on trees just as in section 4. Applying Theorem 5.1 to the instance of `Demand MultiCut` obtained here can be shown to give Theorem 6.1.

6.2. $O(1)$-approximation for `WtdFlowTime` in time polynomial in n, W. Our algorithm for `WtdFlowTime` 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. Denote the maximum and minimum weights, and processing times, of any job in the instance $\mathcal{I}$ of `WtdFlowTime` by $w_{max}, w_{min}, p_{max}, p_{min}$, respectively. The rough idea is as follows. We will choose δ such that $p_{\max}/\delta$ is polynomially bounded. We ignore all jobs of prcessing time less than δ and solve the remaining problem using our algorithm (where P will be polynomially bounded). Now, we reintroduce the jobs of processing time than δ and show that increase in weighted flow-time will be small if the weights are polynomially bounded.

LEMMA 6.4. *There is a constant factor approximation algorithm for* `WtdFlowTime` *where the running time of the algorithm is polynomial in n and W.*

Proof. We begin with a claim about considering jobs of processing time less than δ separately, for a fixed δ that will be determined later. We divide the given instance $\mathcal{I}$ of `WtdFlowTime` into two instances, $\mathcal{I}_{small}$ and $\mathcal{I}_{rem}$. Here, $\mathcal{I}_{small}$ consists of the jobs of processing time less than δ, and $\mathcal{I}_{rem}$ consists of the remaining jobs. We run our

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

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

algorithm from Theorem 1.1 on $\mathcal{I}_{rem}$ to obtain a schedule S_{rem}. We then construct a new schedule S by reintroducing jobs from $\mathcal{I}_{small}$ as follows. We first determine an arbitrary busy schedule S_{small} for the jobs in $\mathcal{I}_{small}$. We then schedule the jobs of $\mathcal{I}_{rem}$ in the slots left free by S_{small} in the same order as S_{rem}. Note that S will not violate release dates.

We now analyze the total weighted flow-time of S, which we denote by $wf(S)$ (analogously for S_{small} and S_{rem}). Also, let OPT denote the optimum weighted flow-time for $\mathcal{I}$. The flow-time of any job in $\mathcal{I}_{small}$ is at most $n\delta$ (because the schedule is busy and all jobs in $\mathcal{I}_{small}$ have processing time at most δ). Also, the increase in flow-time of any job in $\mathcal{I}_{rem}$ in S from its flow-time in S_{rem} is also at most $n\delta$ by the same argument as before. Hence, $wf(S)$ is at most $2n^2\delta w_{\max}$ more than $wf(S_{rem})$. By the property of our algorithm from Theorem 1.1, $wf(S_{rem})$ is at most a constant (call the constant r) times the optimum weighted flow-time for $\mathcal{I}_{rem}$ which is at most OPT. Hence, we have

$$\begin{aligned}
wf(S) &\leq 2n^2\delta w_{\max} + r \cdot OPT && (6.1)\\
&\leq 2n^2\delta w_{\min} W + r \cdot OPT && (6.2)\\
&\leq 2p_{\max} w_{\min}(\delta W n^2/p_{\max}) + r \cdot OPT && (6.3)\\
&\leq OPT(2\delta W n^2/p_{\max}) + r \cdot OPT, && (6.4)
\end{aligned}$$

□

where the last step uses the fact that $OPT \geq p_{max} w_{min}$.

We now choose $\delta = p_{max}/2n^2W$. This ensures that the algorithm is a constant factor approximation by our analysis above. Also, since p_{max}/δ is polynomial in n and W, the algorithm for computing S_{rem} now runs in time polynomial in n, W. Hence, the entire algorithm runs in time polynomial in n, W. This completes the proof.

7. Discussion.

We give the first pseudopolynomial time constant factor approximation algorithm for the weighted flow-time problem on a single machine. The main open question here is whether it is possible to obtain a smaller constant (or even PTAS) for `WtdFlowTime`. It would also be interesting to discover constant integrality gap formulations for `WtdFlowTime`. Further, for `Demand MultiCut`, one could ask if there is a polynomial time $O(1)$ approximation, or even whether the canonical LP for `Demand MultiCut` augmented with knapsack cover inequalities has $O(1)$ integrality gap.

Acknowledgment. The authors are grateful to the anonymous referees for helpful comments.

REFERENCES

[1] Y. Azar and N. Touitou, *Improved online algorithm for weighted flow time*, in Proceedings of the 2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS), 2018, pp. 427–437.

[2] N. Bansal and H. Chan, *Weighted flow time does not admit o(1)-competitive algorithms*, in Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, C. Mathieu, ed., SIAM, 2009, pp. 1238–1244.

[3] N. Bansal and K. Dhamdhere, *Minimizing weighted flow time*, ACM Trans. Algorithms, 3 (2007), p. 39, https://doi.org/10.1145/1290672.1290676.

[4] N. Bansal, R. Krishnaswamy, and B. Saha, *On capacitated set cover problems*, in Approximation, Randomization, and Combinatorial Optimization: Algorithms and Techniques, Springer-Verlag, Berlin, 2011, pp. 38–49.

[5] N. Bansal and K. Pruhs, *Server scheduling in the weighted pnorm*, in LATIN 2004: Theoretical Informatics, M. Farach-Colton, ed., Berlin, Heidelberg, 2004, Springer Berlin Heidelberg, pp. 434–443.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 02/13/24 to 192.16.191.136 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

[6] N. Bansal and K. Pruhs, *The geometry of scheduling*, SIAM J. Comput., 43 (2014), pp. 1684–1698, https://doi.org/10.1137/130911317.

[7] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Schieber, *A unified approach to approximating resource allocation and scheduling*, J. ACM, 48 (2001), pp. 1069–1090, https://doi.org/10.1145/502102.502107.

[8] M. A. Bender, S. Muthukrishnan, and R. Rajaraman, *Approximation algorithms for average stretch scheduling*, J. Scheduling, 7 (2004), pp. 195–222, https://doi.org/10.1023/B:JOSH.0000019681.52701.8b.

[9] D. Chakrabarty, E. Grant, and J. Könemann, *On column-restricted and priority covering integer programs*, in Proceedings of the 14th International Conference, IPCO 2010, Lausanne, Switzerland, 2010, pp. 355–368, https://doi.org/10.1007/978-3-642-13036-6_27.

[10] T. M. Chan, E. Grant, J. Könemann, and M. 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, 2012, pp. 1576–1585.

[11] C. Chekuri and S. Khanna, *Approximation schemes for preemptive weighted flow time*, in Proceedings on 34th Annual ACM Symposium on Theory of Computing, Montréal, Québec, Canada, 2002, pp. 297–305, https://doi.org/10.1145/509907.509954.

[12] C. Chekuri, S. Khanna, and A. Zhu, *Algorithms for minimizing weighted flow time*, in Proceedings on 33rd Annual ACM Symposium on Theory of Computing, Heraklion, Crete, Greece, 2001, pp. 84–93, https://doi.org/10.1145/380752.380778.

[13] U. Feige, J. Kulkarni, and S. Li, *A polynomial time constant approximation for minimizing total weighted flow-time*, in Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, 2019, pp. 1585–1595, https://doi.org/10.1137/1.9781611975482.96.

[14] N. Garg, V. V. Vazirani, and M. Yannakakis, *Primal-dual approximation algorithms for integral flow and multicut in trees*, Algorithmica, 18 (1997), pp. 3–20, https://doi.org/10.1007/BF02523685.

[15] S. Im and B. 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, 2017, pp. 2601–2615, https://doi.org/10.1137/1.9781611974782.171.

[16] B. Kalyanasundaram and K. Pruhs, *Speed is as powerful as clairvoyance*, J. ACM, 47 (2000), pp. 617–643, https://doi.org/10.1145/347476.347479.

[17] H. Kellerer, T. Tautenhahn, and G. Woeginger, *Approximability and nonapproximability results for minimizing total flow time on a single machine*, SIAM J. Comput., 28 (1999), pp. 1155–1166.

[18] J. Lenstra, A. R. Kan, and P. Brucker, *Complexity of machine scheduling problems*, in Studies in Integer Programming, P. Hammer, E. Johnson, B. Korte, and G. Nemhauser, eds., Discrete Ann. Math. 1, Elsevier, New York, 1977, pp. 343–362, https://doi.org/10.1016/S0167-5060(08)70743-X.

[19] K. Varadarajan, *Weighted geometric set cover via quasi-uniform sampling*, in Proceedings of the Forty-second ACM Symposium on Theory of Computing, STOC '10, New York, NY, 2010, ACM, pp. 641–648, https://doi.org/10.1145/1806689.1806777.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.