Minimizing flow time on a constant number of machines with preemption

Nikhil Bansal*

Abstract

We consider offline algorithms for minimizing the total flow time on \(O(1)\) machines where jobs can be preempted arbitrarily but migrations are disallowed. Our main result is a quasi-polynomial time approximation scheme for minimizing the total flow time. We also consider more general settings and give some hardness results.

Keywords: Flow time, approximation schemes, multiple machines, preemption, scheduling.

1 Introduction

Flow time, defined as the time elapsed since a request is submitted until it finishes, is one of the most relevant measures of the quality of service received. A natural problem then is to minimize the total flow time, given a collection of \(n\) jobs (requests) arriving at arbitrary times and having arbitrary processing requirements.

We consider this problem in the preemptive setting where jobs can be interrupted arbitrarily and resumed later from the point of interruption. This is essential, as it is known that achieving an approximation ratio of \(O(n^{1/3-\epsilon})\) is impossible in the non-preemptive case, unless \(P = NP\) [9]. The single machine case with preemptions is well understood. It is known that the Shortest Remaining Processing Time (SRPT) algorithm, that at any time works on the job with least remaining service, is optimal for minimizing the total flow time on a single machine with preemptions [12]. However, for the case of \(m \geq 2\) machines, the problem is NP-Hard [5]. The first non-trivial result for the multiple machines case was an \(O(\log n)\) approximation algorithm due to Leonardi and Raz [11]. The algorithm of [11] involved migration of jobs (i.e. a job can be interrupted on one machine and moved to another). An \(O(\log n)\) approximation algorithm without migrations was given later [2]. While these algorithms work for an arbitrary number of machines, no algorithm with an approximation ratio of \(o(\log n)\) is known even for the case of \(m = 2\) machines. On the other hand, it is not known whether the problem is APX-hard. Obtaining a constant factor approximation algorithm even for the case of \(m = 2\) is a major open problem in scheduling [13].

While elegant approximation schemes are known for other related problems such as minimizing the total completion time on multiple machines [1], or minimizing the makespan [8], not many results are known for flow time. One difficulty in obtaining efficient approximation schemes is that flow time is very sensitive to small changes in the schedule.

In this paper, we give a quasi-polynomial time approximation scheme (QPTAS) for minimizing the total flow time on a constant number of machines with preemption, but without migrations.

*IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598. nikhil@us.ibm.com
Our result builds up on the recent result of Chekuri and Khanna [4], who gave an elegant (QPTAS) for minimizing the weighted flow time on a single machine with preemptions. The results of [4] however, do not extend to the case of \( m > 1 \) in a direct way. The main difficulty is that if we consider a particular machine, an arbitrary subset of jobs could be assigned to be processed on it. As we need to keep track of the state (remaining processing requirements, number of jobs etc.) in sufficient detail it is not clear how to encode this information using few bits.

Our main contribution is a technique that allows us to store the approximate state under SRPT for any subset of jobs using \( O(\log^2 n) \) bits of information (observe that, reducing the number of bits required to \( O(\log n) \) would imply a polynomial time approximation scheme). Moreover, this state description has sufficient information to allow us to compute the new state as time progresses and jobs get worked upon or when new jobs arrive. Thus, for each time \( t \) (we show later we only need to consider \( O(n^3) \) values of time) and for each possible configuration, we can compute the best possible total flow time achievable using dynamic programming. We use this to obtain a \((1 + \epsilon)\) approximation algorithm with running time \( n^{(O(\log n)/\epsilon^2)} \).

We extend the above result to the case of unrelated machines and weighted jobs. In the unrelated machines setting, a job \( J_j \) could have a different service requirement (size) on each machine.

Formally, we give an approximation scheme with running time \( 2^{O(m(\min(\log(nW), \log(nB)))/\epsilon^3)} \) for minimizing the total weighted flow time on unrelated machines with preemptions but no migrations. Here \( W \) (resp. \( B \)) is the ratio of maximum to minimum job weight (resp. size). Prior to our work, only the single machine case had been considered, for which Chekuri and Khanna gave an approximation scheme with a running time of \( n^{(\log B \log W)/\epsilon^3} \) [4].

The running time in the above algorithms depends exponentially on \( m \). We show that this dependence is unlikely to be improved for the weighted case. In particular, we show that, if \( m \) is a part of the input, then an algorithm that has approximation ratio \( n^{o(1)} \) and running time \( n^{O(\text{polylog}(n,m,W,B))} \) for minimizing total weighted flow time on identical machines would imply that \( \text{NP} \subseteq \text{DTIME}(2^{O(\text{polylog}(n))}) \).

Finally, we consider the problem of minimizing the maximum flow time on multiple unrelated machines where migration of jobs is disallowed. It is easy to see that the algorithm First In First Out (FIFO) is optimal for the single machine case. Moreover, by a trivial reduction from the Partition Problem [6], it follows that the problem is NP-hard even for \( m = 2 \) identical machines. The best known result is an online algorithm due to Bender et al. [3], that is \( 3 - 2/m \) competitive for \( m \) identical machines. In the special case when all the release times are 0, minimizing the maximum flow time is identical to minimizing the makespan on multiple unrelated machines, for which a 2-approximation is known [10]. In this paper, we given a \((1 + \epsilon)\) approximation algorithm for minimizing the maximum flow time with general release times that runs in time \( n^{O(m/\epsilon)} \). This implies a PTAS for \( O(1) \) machines. As minimizing the makespan on multiple unrelated machines is APX-hard, if \( m \) is a part of the input [10], this implies that the dependence on \( m \) in the running time of our algorithm cannot be improved substantially.

## 2 Total Flow Time

We first consider the problem of minimizing the total flow time when all machines are identical and all jobs are unweighted.

Our high level idea is the following: We first show that the input instance can be rounded such that all job sizes are integers in the range \([1, n^2/\epsilon]\) and all the release times are integers in the range \([1, n^3/\epsilon]\). We then show how to store the approximate state under SRPT for any subset
of jobs using $O(\log^2 n)$ bits of information. Finally, we show how this implies an approximation scheme for the problem.

Let $I$ be a problem instance and let $B$ denote the largest job size. Without loss of generality we can assume that all release dates are at most $nB$ and that all jobs finish execution by time $2nB$ (otherwise we could reduce the problem into two disjoint problems). Let $Opt$ denote the optimal schedule. We also abuse notation and use $Opt$ to denote the total flow time under the optimal schedule.

**Lemma 1** Given $I$, rounding up the job sizes and release dates to a multiple of $\epsilon B/n^2$ only increases the optimal cost of this rounded instance by a factor of at most $(1 + 2\epsilon)$.

**Proof:** Given a schedule for the original instance, rounding up the job sizes adds at most $n \cdot \epsilon B/n^2 = \epsilon B/n$ to the flow time of each job. Similarly rounding up the release dates adds at most $\epsilon B/n^2$ to the flow time of each job. Thus, the total flow time is affected by at most $2\epsilon B \leq 2\epsilon Opt$. \hfill $\Box$

Let $I'$ denote this rounded instance. By Lemma 1 we can assume that all job sizes are integers in the range $[1, n^2/\epsilon]$. Similarly, as no release date in $I$ is more than $2nB$, and the time is rounded to a multiple $\epsilon B/n^2$, all release dates in $I'$ are integers in the range $[1, n^3/\epsilon]$. Consider a schedule $S(I')$ for $I'$. Clearly, all events (arrivals and departures) under $S(I')$ occur at integral times in the range $[1, 2n^3/\epsilon]$. Also, a schedule $S(I)$ for $I$ follows from $S(I')$ and the total flow time under $S(I)$ is at most that under $S(I')$. To see this, consider $S(I)$ obtained from $S(I')$ by scheduling a job in $I$ only when the corresponding job in $I'$ is scheduled. If the job in $I$ finishes before the corresponding job in $I'$ (since the job in $I'$ could be larger), then the scheduler idles for that time. Finally, since the release time for a job in $I'$ is no earlier than the corresponding job in $I$, the schedule produced thus is a feasible schedule for $I$. Henceforth, we only consider $I'$.

We now describe how to compute an approximately optimal schedule (i.e. up to a factor of $1 + O(\epsilon)$) for $I'$. We say that a job with size $p_j$ lies in class $i$, iff $(1 + \epsilon)^{i-1} \leq p_j < (1 + \epsilon)^i$. Since all the sizes are in the range $[1, n^2/\epsilon]$. This divides the jobs into $\log_{(1+\epsilon)}(n^2/\epsilon) = O(\log_{(1+\epsilon)} n^2) = O(\frac{1}{\epsilon} \log n)$ classes. Note that the class of a job depends only on its initial processing time and hence does not change with time. Consider the optimum algorithm. Let $S_k(i, t)$ denote the set of jobs of class $i$ on machine $k$ which are alive at time $t$. The lemma below gives an important property of the optimal schedule.

**Lemma 2** For all $k = 1, \ldots, m$ and at all times $t$, at most one job in $S_k(i, t)$ has remaining processing time that is not in the range $(1 + \epsilon)^{t-1}$ and $(1 + \epsilon)^t$.

**Proof:** As the shortest remaining processing time (SRPT) is optimal for minimizing the total flow time on a single machine, we can assume that on each machine, the jobs assigned to it are processed in the SRPT order.

Let us consider the class $i$ jobs on a machine $k$. Since at any time we execute the job with least remaining time, if a job from this class is executed, clearly it must be the one that has the least remaining time in $S_k(i, t)$. Suppose for the sake of contradiction that there are two jobs, $J_1$ and $J_2$, with remaining time less than $(1 + \epsilon)^{t-1}$ at time $t$. Consider the time $t'$ when $J_1$ was worked upon and its remaining processing time first became less than $(1 + \epsilon)^{t-1}$. If $J_2$ had remaining time less then $(1 + \epsilon)^{t-1}$ at $t'$, then $J_1$ should not have been worked upon (according to SRPT). On the other hand, if $J_2$ has not arrived by $t'$ or its remaining time was greater than $(1 + \epsilon)^{t-1}$ at $t'$, then according to SRPT $J_2$ should not have been worked upon at any time during $t'$ and $t$, which contradicts that remaining processing time of $J_2$ at time $t$ is less than $(1 + \epsilon)^{t-1}$. \hfill $\Box$
For an algorithm, we define its state \( Q(t) \) at time \( t \) as follows: For each class \( i \) and each machine \( k \), we store \((\frac{1}{\epsilon} + 1)\) numbers: the first \( \frac{1}{\epsilon} \) are the remaining processing times of the \( \frac{1}{\epsilon} \) jobs in \( S_k(i, t) \) with the largest remaining processing time; the last entry is the sum of the remaining processing times of the rest of the jobs in \( S_k(i, t) \). Notice that as each job has size at most \( n^2/\epsilon \), there are at most \((n^2/\epsilon)^{1/\epsilon} = O(n^{2/\epsilon})\) possible choices for the first \( 1/\epsilon \) entries and \( n^{3/\epsilon} \) possible choices for the sum of remaining sizes. Finally, since there are at most \( O(\frac{1}{\epsilon} \log n) \) classes and \( m \) machines, the total number of distinct states at any step is at most \( n^{O(1/\epsilon \cdot m \cdot \frac{1}{\epsilon} \log n)} = n^{O((m \log n)/\epsilon^2)} \).

The following lemma shows how this information helps us in estimating the number of unfinished jobs at any point of time.

**Lemma 3** We can estimate the number of jobs in the system at time \( t \) to within a factor of \((1 + 2\epsilon)\) using the information in \( Q(t) \).

**Proof:** If there are fewer than \( 1/\epsilon \) jobs in level \( i \), we know their number precisely because we store their remaining processing times precisely. Let us now consider the case when there are more than \( 1/\epsilon \) jobs in some level \( i \).

Suppose at first that the remaining processing time of all these jobs lies between \((1 + \epsilon)^{i-1}\) and \((1 + \epsilon)^i\). Then, by assuming that all the jobs have size \((1 + \epsilon)^{i-1}\) and computing the number of jobs using the total remaining processing time, our estimate is off from the correct number by at most a factor of \((1 + \epsilon)\).

Finally, as at most one job in every level \( i \) lies outside the range \((1 + \epsilon)^{i-1}\) and \((1 + \epsilon)^i\). Thus our estimate above could be off by another job. However, as there are at least \( 1/\epsilon \) unfinished jobs, our estimate is within a factor of \((1 + 2\epsilon)\). \(\square\)

Thus the algorithm to compute the approximately optimal schedule will be a large dynamic program which will have a state for each time unit and each of the possible configurations of \( Q(t) \) at that time. Thus there will be at most \( O(n^3/\epsilon)\) times \( n^{O((m \log n)/\epsilon^2)} \) states. As usual, with each state we store the value of the least total flow time incurred to reach that state.

To complete the description of the dynamic program, we only need to show how to update the state of the algorithm with time. When a new job arrives, we have \( m \) choices for the machine to which this job can be assigned. Once a machine is decided, the size of the job determines the class to which it belongs. Also, it is straightforward to update the state, as either the job is added to the first \( 1/\epsilon \) jobs in \( S_k(i, t) \), or else if it is smaller than the \( 1/\epsilon \) largest jobs in its class, then its size is added to the \((1/\epsilon + 1)^{th}\) entry. Now consider the case when there are no arrivals. When the algorithm works on a job in level \( i \) on machine \( k \), if there are \( 1/\epsilon \) or fewer jobs in level \( i \), then we simply decrement the smallest of the remaining time entries by \( 1 \). Else, if there are more than \( 1/\epsilon \) jobs, we decrement the total remaining processing time entry by \( 1 \). In general, we will not know in which level the job with the least remaining processing time is present (as we do not have this information in our state description). However, we can try out all possible \( O((\log n)/\epsilon)\) choices for the different levels for each machine \( k \). Thus, we have at most \( O(\log n/\epsilon)^m \) choices at each time step and for each possible state. Thus the total running time at each time step is \( O((\log n)/\epsilon)^m \cdot n^{O((m \log n)/\epsilon^2)} = n^{O((m \log n)/\epsilon^2)} \).

Finally, having computed the approximately optimal value of flow time, it is straightforward to compute the path used to reach this value, which gives us the \((1 + 2\epsilon)\) optimal schedule for \( T' \). Finally by Lemma 1 this implies that

**Theorem 1** The above algorithm gives a \((1 + \epsilon)\) approximation for minimizing the total flow time on \( m \) identical machines and runs in time \( n^{O((m \log n)/\epsilon^2)} \).
3 Extensions

We now extend the algorithm for the more general case of unrelated machines and weighted jobs. In the unrelated machines setting, a job $J_k$ is described by an $m$-tuple $(p_{j1}, \ldots, p_{jm})$, where $p_{jk}$ denotes its size on machine $k$. Let $p_{j}^*$ denote $\min_k p_{jk}$. Let $Q = \sum_j p_{j}^*$, and suppose that the weights are integers in the range $1, \ldots, W$. For unrelated machines, we define $B = \max_{i \neq j} p_{i}^*/p_{j}^*$. Note that if the machines are identical, this definition of $B$ corresponds exactly to the maximum of minimum job size ratio. Also it is easily seen that $Q \leq nB \min_j p_{j}^*$.

Our algorithm will consist of two different algorithms for different cases. First, for the case when $W \leq B$, we give an algorithm with running time $2^{O((m \log^3(nW))/\epsilon^3)}$. For the case when $W \geq B$, we give another algorithm with running time $2^{O((m \log^3(nB))/\epsilon^3)}$. Combining these will give our final algorithm with running time $2^{O((m \min \log(nB), \log(nW))/\epsilon^3)}$.

We first consider the case when $W \leq B$. By assigning each job to the machine where it takes the least time, it is clear that $nWQ$ is an upper bound on $Opt$. Similarly, $Q$ is a lower bound on $Opt$. Hence we will assume that our algorithm simply never assigns a job $J_j$ to machine $k$ if $p_{jk} \geq 2nWQ$. Imitating the proof of Lemma 1, it easily follows that rounding each $p_{ij}$ and release date up to the next multiple of $\epsilon Q/(Wn^2)$ affects the flow time of each job by at most $\epsilon Q/Wn$. Since the maximum weight of any job is at most $W$, this affects the total weighted flow time by at most $\epsilon Q \leq \epsilon Opt$. Moreover, we can assume that the release date of any job is at most $2nQ$ (otherwise the problem can be decomposed into two disjoint instance). Thus after rounding the release dates and sizes to multiples of $\epsilon Q/(Wn^2)$, there are only $O(n^2W/\epsilon)$ time steps to consider. Next we round up the weights to powers of $(1 + \epsilon)$. Clearly, this affects the solution by a factor of at most $(1 + \epsilon)$. Finally, observe that in the optimal schedule for this rounded instance, if we consider a particular machine and restrict our attention to time intervals when jobs from a particular weight class are executed, then these jobs are executed in the SRPT order.

With these observations we can directly give an algorithm based on the ideas in Section 2. For each machine and each weight class, we maintain the possible states under an SRPT schedule as in Section 2. Since there are $O((\log W)/\epsilon)$ weight classes and $O(\log(nW)/\epsilon)$ size classes, and for each size class we store $O(\log(nW)/\epsilon)$ bits of information, the number of states at each time step is bounded by $2^{O((m \log^3(nW))/\epsilon^3)}$. When a job arrives, there are $m$ choices corresponding to the machines it can be assigned. If there are no arrivals, for each machine, we need to decide which weight class to work on, and within each weight class the size class to work on. Thus there are $(O(\log^2(nW))/\epsilon^2))^m$ choices to choose from at each time step. Finally, as all the release dates and sizes are integers between 1 and $O(n^2W/\epsilon)$, our algorithm can be implemented directly as a dynamic program of size $\epsilon W^3/n$ times $2^{O((m \log^3(nW))/\epsilon^3)}$. This gives the desired bound on the running time.

We now consider the case when $B \leq W$. As previously, $Opt$ is at most $nQW$. Since $Q \leq nB \min_i p_i^*$, it follows that each job has size at least $Q/nB$ on each machine. Finally, since the maximum weight is $W$, it follows that $Opt$ is also lower bounded by $WQ/nB$.

Our algorithm now is as follows. We only consider jobs with weights between $\epsilon W/n^2B$ and $W$. Jobs with weight less than $\epsilon W/n^2B$ will be added to the schedule arbitrarily. Since each job has flow time at most $Q$, this adds at most $\epsilon WQ/nB \leq \epsilon Opt$. Similarly, rounding up the job sizes and release dates to a multiple of $\epsilon Q/n^3B$ affects the total weighted flow time by at most $n^2W \cdot \epsilon Q/n^3B = \epsilon WQ/nB \leq \epsilon Opt$. Thus after rounding we have an instance where the job sizes and release dates are integers between 1 and $O(n^2B)$ and the job weights are in the range $W/n^2B$ and $W$. Thus there are $O(\log(nB)/\epsilon)$ weight classes and $O(\log(nB)/\epsilon)$ sizes classes. Applying the algorithm described above for the case $W \leq B$ and observing that all release dates are bounded
by \( nB \), we get that the running time of the algorithm is \( 2^{O(m\log^3(nB)/\epsilon^3)} \).

Thus we have shown that

**Theorem 2** There is an algorithm for minimizing the total weighted flow time on multiple unrelated machines that produces a \((1+\epsilon)\) approximate solution and runs in time \( 2^{O(m\min\{\log(nB),\log(nW)\}^3/\epsilon^3)} \).

## 4 Dependence on the number of machines

It is known that minimizing the total unweighted flow time or minimizing the maximum flow time on unrelated machines are both APX-hard for arbitrary \( m \) [7, 10]. However, these results do not rule out the possibility of an approximation scheme with a polynomial dependence on \( m \) for identical parallel machines. We now show that minimizing total weighted flow time on \( m \) identical machines is APX-hard, if \( m \) part of the input.

Consider an instance of 3-Partition (SP15) in [6]. This consists of a set \( A \) of 3\( m \) elements, an integer bound \( S > 0 \); for each \( x \in A \) a integer size \( s(x) \) s.t. \( S/4 < s(x) < S/2 \) and s.t. \( \sum_{x \in A} s(x) = mS \). The question is whether \( A \) can be partitioned into \( m \) disjoint sets \( A_1, A_2, \ldots, A_m \) such that, for each \( 1 \leq i \leq m \), \( A_i \) has 3 elements and \( \sum_{a \in A_i} s(a) = S \). 3-Partition is known to be strongly NP-complete. In particular, it is NP-complete for \( S = O(m^4) \).

Given an instance of 3-Partition, we transform it as follows. There are \( m \) machines. Each element \( x \in A \) corresponds to a job, with size \( s(x) \), weight and is released at time \( t = 0 \). Next at each time instance \( t = S + i/m^2 \), for \( i = 0, 1, 2, \ldots, Sm^5 \), \( m \) jobs each with size \( 1/m^2 \) and weight \( 1/m \) are released. Thus the total number of these small jobs is \( Sm^6 \).

If the instance of 3-partition has a solution, then we construct a schedule as follows: We schedule the jobs that arrive at \( t = 0 \) according to the solution to the instance of 3-partition. Since each job finishes by time \( t = S \), we can schedule the jobs of weight \( 1/m \) as they arrive. Now, each of the weight \( m \) job has flow time at most \( 3S \), and each of the \( Sm^6 \) jobs of weight \( 1/m \) has flow time \( 1/m^2 \). Thus the total weighted flow time is \( 3m \cdot 3S \cdot m + 1/m^3 \cdot Sm^6 = O(Sm^3) \).

On the other hand if there is no solution to the 3-partition instance, there is at least one weight \( m \) job (call it \( J \)) unfinished by time \( S \). In particular, the job \( J \) has at least one unit of work unfinished by time \( S \). Consider the situation by time \( Sm^3/2 \), if \( J \) is still there, it contributes at least \( Sm^4/2 \) to the flow time, else there are at least \( m^2 \) jobs of size \( 1/m^2 \) and weight \( 1/m \) piled up during the time interval \( [Sm^3/2, Sm^3] \), thus contributing \( O(Sm^4) \) to the total weighted flow time.

As \( S = O(m^4) \), \( W = m^2 \) and \( n = Sm^6 = O(m^{10}) \), it follows that \( W \) and \( S \) are polynomially bounded in \( n \) and the inapproximability factor of \( \Omega(m) \) implies an inapproximability factor of \( \Omega(n^{1/10}) \). Thus,

**Theorem 3** It is NP-hard to obtain an \( n^{o(1)} \) approximation for minimizing the total weighted flow time on \( m \) identical machines if \( m \) is part of the input, even if \( W \) and \( B \) are polynomially bounded in \( n \).

As a corollary, we also obtain that,

**Theorem 4** Any \( n^{o(1)} \) approximation algorithm for minimizing the total weighted flow time on multiple identical machines that runs in time \( 2^{\text{polylog}(n,m,W,B)} \) would imply that \( NP \subseteq \text{DTIME}(2^{\text{polylog}(n)}) \).
5 Maximum Flow Time

We now consider the problem of minimizing the maximum flow time on $m$ unrelated machines, where the jobs cannot be migrated from one machine to another. We give an algorithm that runs in time $n^{O(m/\epsilon)}$ and produces a $(1 + \epsilon)$ approximation. We begin with some easy observations.

1. As FIFO is optimal for minimizing the maximum flow time on a single machine. It follows that for every machine, the jobs assigned to that machine are executed in FIFO order.

2. As in Section 3, let $p^*_j$ denote $\min_k p_{jk}$ and $Q = \sum_j p^*_j$. By assigning each job to the machine where it takes the least time, it is clear that $Q$ is an upper bound on Opt. Furthermore, we can assume that all the release dates are at most $Q$ and that all jobs finish execution by time $2Q$ (otherwise, the problem can be reduced to two disjoint problem instances). Finally, as there are $n$ jobs, there is some $j$ such that $p^*_j \geq Q/n$ and hence this job has size at least $Q/n$ on every machine. Thus $Q/n$ is a lower bound on Opt.

3. Let $\delta > 0$ be an arbitrary real number. Note that rounding up the release date of each job to the next multiple of $\delta$ increases the maximum flow time by at most $\delta$. Similarly, rounding up each $p_{ij}$ to the next multiple of $\delta$ can increase the maximum flow time by at most $n\delta$.

We now describe the algorithm. If $p_{jk} > 2Q$ for some $j, k$, we just set $p_{jk} = \infty$. Choose $\delta = \epsilon Q/4n^2$ and round each $p_{jk}$ and $r_j$ up to the next multiple of $\delta$. As all job sizes and release dates are multiples of $\delta$, we can discretize time in steps of $\delta$ and hence there are only $O(n^2)$ time steps. The algorithm maintains the state $(Val, t, Z_1, Z_2, \ldots, Z_m)$ where $t$ denotes the time, $Z_k$ represents the total unfinished work present on machine $k$ at time $t$, and $Val$ is minimum value of the maximum flow time that can be achieved if no new jobs arrive.

It is straightforward to update the state of the algorithm as follows: If there is a new arrival $J_i$ at time $t$, the algorithm explores each of the $m$ choices for the machine where the job is assigned. When the algorithm assigns job to machine $k$, it updates the state as $Z_k = Z_k + p_{ik}$ and updates $Val = \max_k \{Val, Z_k + p_{ik} \}$. If no jobs arrives at time $t$, the algorithm increments $t$ to $t + 1$ and decrements $Z_k$ to $Z_k - 1$ for each $Z_k > 0$. As there are at most $O(n^3/\epsilon)$ possible choices for each of $Val$, $t$ and $Z_1, \ldots, Z_m$, there are at most $n^{O(m/\epsilon)}$ states. Thus,

**Theorem 5** There exists a $(1 + \epsilon)$ approximation algorithm with running time $n^{O(m/\epsilon)}$ for minimizing the maximum flow time on $m$ unrelated machines without migration.

6 Conclusions

Our result suggests that a polynomial time approximation scheme is likely for the problem of minimizing total flow time on a constant number of machines.

We are unable to extend our results to the case when jobs can be migrated from one machine to another. In the non-migratory case, storing the aggregate information about jobs suffices to obtain a good estimate of the number of jobs at any time. In particular, our algorithm maintains no information about the remaining processing time on a partially executed job. In the migratory case, as we can move partially executed jobs from one machine to another, it is not clear how to keep track of the global state without keeping track of the remaining service on individual jobs. The latter can cause the number of states to become exponentially large.
References


