New Results for Lazy Bureaucrat Scheduling Problem

Arash Farzan Mohammad Ghodsi
{a_farzan@ce., ghodsi@}sharif.edu

Computer Engineering Department
Sharif University of Technology

Oct. 10, 2001

Abstract

Lazy bureaucrat scheduling is a new class of scheduling problems that was introduced in [1]. In this class of scheduling problems, there is one employee (or more) who should perform the assigned jobs; the objective of the employee is to minimize the amount of work he does and be as inefficient as possible (he is "Lazy"). He is subject to a constraint that he should be busy when there is work to be done. In this paper, we first briefly define the "lazy bureaucrat scheduling" model as it is introduced in [1] and present some old results. We then present our extensions and new results on this problem. We show that when all jobs have unit length, the latest deadline first policy minimizes the amount of executed work both in a single processor and multiple processors; so optimal schedule can be found in polynomial time. We also prove that if all jobs have a common release time and the objective function is to minimize the weighted sum of completed jobs, the optimum schedule can be found in polynomial time.

Keywords:
Unit-length Jobs, Narrow Window, Lazy Bureaucrat Problem, Lazy Bureaucrat Scheduling.

1 Introduction

Scheduling problems have been studied extensively from the point of view of employers (e.g. see [2]). In these problems the objective is to perform the jobs as efficient as possible and to maximize the amount of completed jobs. We take a new look at this problem from the point of view of the employees who should do the jobs. It is quite natural to expect that some employees may lack the motivation to perform their jobs efficiently. We call such employees "lazy". The following example illustrates such case for a typical office worker:

> **Example [1].** It is 3:00 p.m., and Dilbert goes home at 5:00 p.m. Dilbert has two tasks that have been given to him: one requires 10 minutes, the other requires an hour. If there is a task in his "in-box", Dilbert must work on it, or risk getting fired. However, if he has multiple tasks, Dilbert has the freedom to choose which one to do first. He also knows that at 3:15, another task will appear: a 45-minute personnel meeting. If Dilbert begins the 10-minute task first, he will be free to attend the personnel meeting at 3:15 and then work on the hour-long task from 4:00 until 5:00. On the other hand, if Dilbert is part way into the hour-long job at 3:15, he may be excused from the meeting. After finishing the 10-minute job by 4:10, he will have 50 minutes to twiddle his thumbs, iron his tie, or enjoy engaging in other mindless trivia. Naturally, Dilbert prefers this latter option.

This example illustrates a general and natural type of scheduling problem, which is called "Lazy Bureaucrat Problem (LBP)", in which the goal is to schedule jobs as inefficiently as possible. These problems provide an interesting set of algorithmic questions, which may also lead to discovery of structure in traditional scheduling problems. Several other combinatorial optimization problems have also been studied in reverse, leading, e.g. to maximum TSP, maximum cut and longest path.

1.1 The Model

Consider a set of jobs $1 \ldots n$ having processing times (lengths) $t_1 \ldots t_n$ respectively. Job i arrives at time a_i and has its deadline at time d_i. (t_i, a_i, d_i are nonnegative integers). The jobs have hard deadlines and each job i can only be executed during its allowed interval $I_i = [a_i, d_i]$; we also call I_i the job's window.

We study the case in which preemption of jobs is forbidden. A job is preempted if its execution is interrupted and resumed later. So in this paper we suppose that when a job is begun, it must be completed without interruptions. (See [1] for some results when preemption is allowed). We also restrict ourselves to off-line scheduling in which all the jobs are known to the scheduler beforehand. If not said otherwise, we suppose that there is only one processor (or worker) to do the jobs.

Greedy Requirement. The bureaucrat chooses a subset of jobs to execute. Since his goal is to minimize his effort, he prefers to remain idle all the time and to leave all the jobs unexecuted. However, this scenario is forbidden by what we call the *greedy requirement*, which requires that the bureaucrat works on an executable job, if there is any such job. A job is "executable" if it has arrived, its deadline has not yet passed, and it is not yet fully processed.

Objective Functions. In traditional scheduling problems, if it is impossible to complete the set of all jobs by their deadline, one tries to optimize according to some objective , e.g., to maximize a weighted sum of on-time jobs, to minimize the maximum lateness of the jobs, or to minimize the number of late jobs. For the LBP, three different objective functions can be defined ([1]):

1. *Minimize the total amount of time spent working.* This objective naturally appeals to a "lazy" bureaucrat.

2. *Minimize the weighted sum of completed jobs.* This objective appeals to a "spiteful" bureaucrat whose goal is to minimize the fees that the company collects on the basis of his labors, assuming that the fee is collected only for those tasks that are actually completed.

3. *Minimize the makespan, the maximum completion time of the jobs.* This objective appeals to an "impatient" bureaucrat, whose goal is to go home as early as possible, at the completion of the last job, he is able to go. He cares about the number of hours spent at the office, not the number of hours spent doing work (productive or otherwise) at the office.

1.2 Our Results

As it is stated in [1] LBP is strongly NP-complete and is not approximable to within any fixed factor. Thus, we mainly focus on special cases to study algorithms. When all jobs have unit length, optimal schedule can be find in polynomial time. We show that even in the case of more than one processor (multiprocessors) it remains polynomial. Under second objective function in which we have a spiteful worker whose goal is to minimize the weighted sum of completed jobs, when all jobs have a common release time the unit-length LBP can be solved in polynomial time. And finally when each job i's interval I_i is less than twice the length of the job, there is a pseudo-polynomial algorithm; in [1] this algorithm has time complexity of $O(nKmax(n, K))$, when n is the number of jobs and K is the maximum deadline ($K = \max_i d_i$). Here we present an algorithm of complexity $O(nK)$.

In section 2 it is shown that LBP in its general case is NP-complete. In section 3 we discuss about the special cases. All of our new results are in this section. We study two special cases: (1) when all jobs have unit length, which we study thoroughly in subsection 3.1 and (2) when each job i's interval I_i is less that twice the length of i, which we study in subsection 3.2.

2 Hardness Results

In this section we, first, describe the relationship among three different objective functions. The problem of minimizing the total executed work is a special case of the problem of minimizing the weighted sum of completed jobs, because every job that is executed must be completed; in fact, we can define the weights the same as job lengths. Furthermore, if all jobs have the same arrival time, say time zero, then the two objectives, minimizing total executed work and minimizing the makespan (going home as early as possible) are equivalent, since no feasible schedule will have any gaps. Therefore, if we show that the version of LBP problem in which arrival times are all the same and the objective function is to minimize the amount of executed work, is NP-complete, we have also shown that the LBP with any of three objective functions is NP-complete.

Theorem 1 **([1])** *The Lazy Bureaucrat Problem is (weakly) NP-complete, and is not approximable to within any fixed factor, even when arrival times are all the same.*

Proof. We can use a reduction from the subset sum problem [3]. For the complete proof see [1]. □

It can be shown that the problem from Theorem 1 has a pseudo polynomial-time algorithm [1]. However, if arrival times and deadlines are arbitrary integers, the problem is strongly NP-complete. The given reduction applies to all three objective functions.

Theorem 2 **([1])** *The Lazy Bureaucrat Problem is (strongly) NP-complete, and is not approximable to within any fixed factor.*

Proof. We can use a reduction from the 3-partition problem [3]. For the complete proof see [1]. □

3 Algorithms for Special Cases

As we saw in the previous section the LBP problem is NP-complete. So, we can focus on special cases to study algorithms. In this paper, we study two special cases: (1) when all jobs have unit length and (2) when each job i's interval I_i is less than twice the length of i (job i has a narrow window).

3.1 Unit-Length Jobs

Throughout this subsection we consider the special case of the LBP in which all jobs have unit processing times.(Recall that all inputs are assumed to be integral.)

The Latest Deadline First (LDF) scheduling policy selects the job in the system having the latest deadline. In the following theorem we will show LDF policy minimized the amount of executed work. Thus, it makes the unit-length jobs LBP polynomially solvable. We quote the proof completely from [1] as one of our results is based on the proof of this theorem.

Theorem 3 **([1])** *The Latest Deadline First (LDF) scheduling policy minimizes the amount of executed work.*

Proof. Assume by contradiction that no optimal schedule is LDF. We use an exchange argument. Consider an optimal (non-LDF) schedule that has the fewest pairs of jobs executed in non-LDF order. The schedule must have two neighboring jobs i, j such that $i < j$ in the schedule but $d_i < d_j$, and j is in the system when i starts its execution. Consider the first such pair of jobs. There are two cases:

1. The new schedule with i and j switched, is feasible. It executes no more work that optimal schedules, and is therefore also optimal.

2. The schedule with i and j switched is not feasible. This happens if i's deadline has passed. If no job is in the system to replace i, then we obtain a better schedule than the optimal schedule and reach a contradiction. Otherwise, we replace i with the other job and repeat the switching process.

We ultimately obtain a schedule executing no more work than an optimal schedule, with fewest pairs of jobs in non-LDF order, a contradiction. □

Multiple Processors

In the previous theorem we assumed that there is only one processor (or worker) to do the assigned jobs. As an extension, in the following theorem we show that for multiple identical processors the case is quite the same, and it is polynomially solvable.

The LDF policy can also be used to schedule jobs for n processors; consider an arbitrary order $1 \ldots n$ of processors. At each time $(0, 1, 2, \ldots)$ the LDF policy selects the job in the system having the latest deadline for processor 1, the job having the second latest deadline for processor 2 and so on, until there is no executable job in the system.Then, the algorithm increases the time and repeats the process.

The same proof as we used in the previous theorem can prove the correctness of the policy for multiple processors. In fact, the same "exchange argument" is valid for multiple processors.

Theorem 4 *When there is more than one processor, the Latest Deadline First (LDF) scheduling policy minimizes the amount of executed work.* □

Common Release Time

In the next version of the problem all jobs are released at time zero, i.e. $a_i = 0$ for all i and the objective function is to minimize the weighted sum of completed jobs. More precisely, each job i is assigned a weight w_i, if we denote the set of completed jobs by S, we want to minimize $W = \sum_{i \in S} w_i$. As we saw, this objective function may appeal to a spiteful worker whose goal is to minimize the amount of money his company earns. We will show that this version of the problem can be solved in polynomial time.

Theorem 5 *In the version of the unit-length LBP in which all jobs have a common release time, weighted sum of the jobs can be minimized in polynomial time.*

Proof. Since all jobs have a common release time, each eligible schedule of the jobs has the property that it has no gap (a period of idleness) in it; at time zero when all jobs are released, the worker begins working incessantly until some time (say t) and after that time the worker is idle all the time. We call t the makespan of the schedule.

We will show that we can restrict our problem to a fixed makespan without loss of generality. In this new version of the problem we are given a fixed makespan t, we are to find among all the eligible schedules with the makespan of exactly t, the schedule with the least weighted sum of completed jobs.

It is clear that if we can solve the restricted version of the problem in polynomial time, we can also solve the original problem in polynomial time; there is no gap in the schedule so the makespan can be at most n, when n is the number of jobs; to solve the original problem we can repeat the restricted version for $t = 1, 2, \ldots, n$ and declare the least one as the answer for the original problem.

Thus we will focus on the restricted version:

Lemma 1 *Optimum schedule for the restricted version of the problem in Theorem 5 in which the fixed makespan t can be obtained in polynomial time.*

The minimum weighted sum of the restricted version can be find using "Minimum Weight Perfect Matching":

Definition. *Minimum Weight Perfect Matching:* Given a bipartite graph on two sets of vertices A and B and an edge set $E \subseteq A \times B$, a matching is a set of edges whose endpoints contains each vertex of A and B at most once. Suppose that A has no more vertices that B, we call a matching *perfect*, if every vertex of A is in some matching edge. It is also possible to assign weights to the edges, and define the weight of a matching to be the sum of the weights of the matching edges. The key fact that we use in this section is that minimum weight perfect matching can be computed in polynomial time[4].

We will construct a bipartite graph in which a minimum weight perfect matching is equivalent to an optimum schedule for the restricted version of the LBP in Lemma 1.

In the set A we put one vertex for each job so that there are n vertices $A_1, A_2, \ldots, A_n$ and A_i represents job i. Vertices of set B represent unit time intervals; if we define b to be minimum of n and maximum deadline (i.e. $b = \min(n, \max_{i \in \{1,\ldots,n\}} d_i)$), we can be sure that in no eligible schedule makespan is more that b, since no job can be executed after the maximum deadline and as we said previously, no job can be executed after time n. There are b vertices $(B_1, B_2, \ldots, B_b)$ in B each representing a unit-time interval (i.e. B_i represents interval $[i-1, i]$).

Thus far we have defined the vertices of the bipartite graph; now we define the edges: Our bipartite graph is complete and there is an edge between every pair A_i, B_j. For each vertex A_i and each vertex B_j, when $1 \le j \le t$ the weight of the edge (A_i, B_j) is defined to be w_i if $j \le d_i$ and $+\infty$ otherwise. When $t < j \le b$, , weight of the edge (A_i, B_j) is defined to be 0 if $d_i \le t$ and $+\infty$ otherwise. Figure

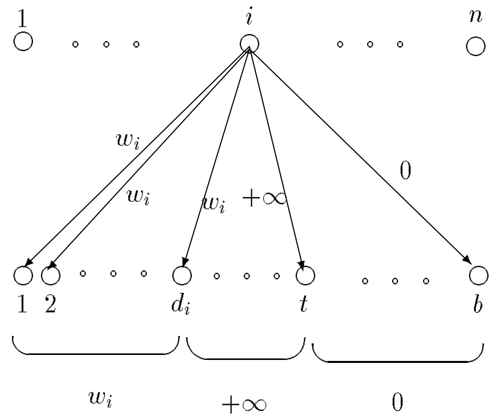

Figure 1: Bipartite graph in case $d_i \leq t$

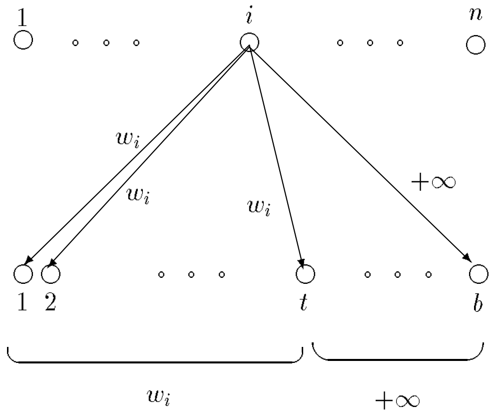

Figure 2: Bipartite graph in case $d_i > t$

1 illustrates the bipartite graph in the case that $d_i \leq t$ and figure 2 illustrates the bipartite graph in the case that $d_i > t$.

We can find the minimum weight perfect matching in this graph in polynomial time. The obtained matching can represent a schedule: If vertex A_i is matched to vertex B_j and if $j \leq d_i$, job i is scheduled at time j; but if $j > d_i$ no job is scheduled at time j. Since a matching must cover the vertices of B, for each time interval B_j, whether a job is assigned or the worker has to be idle during this interval.

We show that the minimum weight perfect matching is equivalent to an optimum schedule. Each schedule with makespan of t represents a finite weight (less than $+\infty$) matching in the graph; if job i is scheduled at time j we match the vertex B_j to the vertex A_i and it is clear that in this matching no edge of weight $+\infty$ is used; since each job in the schedule is performed before its deadline.

On the other hand, each finite weight matching in the graph represents a schedule with makespan t; if edge (A_i, B_j) is in the matching, we schedule job i to be executed at time j. We had assumed that the matching has a finite weight, thus $j < d_i$ and at time j job i is executable. If deadline of a job (say r) is after t ($d_r > t$), job r is matched to a vertex B_j with $j \leq t$; otherwise it must be matched to a vertex B_j with $j > t$ and by definition weight of such edge is $+\infty$ which is a contradiction. Thus, all the jobs with deadline after t are performed before time t, but this means that after time t there is no job to be executed or in other words makespan is t.

Each finite weight matching with weight W is equivalent to an eligible schedule with makespan t and weighted sum W. Therefore, the minimum weight perfect matching is equivalent to the minimum weighted sum schedule of the jobs. $\square$

3.2 Narrow Windows

We now consider the version in which jobs are large in comparison with their intervals, that is, the intervals are "narrow". In this section, by narrow we mean that for each job i, $d_i - a_i < 2t_i$. (See [1] for some other results when the ratio of window length to job length is more that two.)

Let $K = \max_i d_i$ and n be the number of jobs. A pseudo-polynomial algorithm of $O(nK \max(n, K))$

time is presented in [1]. Here we present a pseudo-polynomial algorithm of $O(nK)$ time.

Theorem 6. *Suppose that for each job i, $d_i - a_i < 2t_i$. Then the LBP can be solved in $O(nK)$ time.*

Proof. We use dynamic programming. Let A_t be the minimum amount of time that can be spent working if we start at time t. We will show how A_t can be computed recursively. We fill vector A inversely; that is, at first A_K is computed (it is obviously zero), and after that $A_{K-1}, A_{K-2}, \ldots, A_0$ are computed respectively. It only remains to show that how A_t is recursively computed on the basis of previously computed A_{t_i} $(t_i > t)$.

At time t there may be some jobs that can be executed, that is, t is in their interval. If a job (say j) can be executed at time t, it is impossible that this job had been executed some time before t; if job j can be executed both before and after time t, its interval must be at least twice the length of the job $(d_i - a_i \geq 2t_i)$ which is impossible.

To obtain A_t we consider two cases:

1. There is no executable job at time t. In this case it is obvious that $A_t = A_{t+1}$
2. There are some jobs (say $j_1, j_2, \ldots, j_m$) that are executable at time t. As said previously, none of these jobs can be executable after t, at time t' $(t' > t)$, so we can safely schedule each of them at time t and find the optimum schedule for the rest of time. A_t will be the minimum of all such schedules. More precisely, $A_t = \min_{i \in \{1,2,\ldots,m\}} (t_{j_i} + A_{t+t_{j_i}})$

Analysis of the time complexity of this algorithm is as follows; in the end of the algorithm all values $(A_1, \ldots, A_K)$ of the vector A are computed. To compute each value of the vector, we have to compute the minimum of n values (in the worst case). Thus, this algorithm has time complexity of $O(nK)$. □

4 Conclusion

In this paper we studied several versions of the lazy bureaucrat scheduling problems. In this new class of scheduling problems there is a lazy worker whose main objective is to be as inefficient as possible, in contrast to traditional scheduling problems in which the main objective is to be as efficient as possible.

We saw that the LBP in its general case is strongly NP-complete and hard to approximate. Therefore, we focused on the special cases in which there is some chance for the problem not to be so "hard". We considered two special cases: when all jobs have unit processing time and when each job's window length is less than twice the job length.

When all jobs have unit length, the latest deadline first policy minimizes the amount of executed work both in a single processor and multiple processors; so optimal schedule can be found in polynomial time. We also showed that if all jobs have a common release time and the objective function is to minimize the weighted sum of completed jobs, the optimum schedule can be found in polynomial time.

When each job's window length is less than twice the job length, the optimum schedule can be find in pseudo-polynomial time. We improved the time complexity of this algorithm form $O(nK \max(n, K))$ to $O(nK)$ when n is the number of jobs and K is the maximum deadline.

References

[1] E. M. Arkin, M. A .Bender, J. S. B. Mitchell, S. S. Skiena, *The Lazy Bureaucrat Scheduling Problem*, Workshop on Algorithms and Data Structures, August 1999.

[2] D. Karger, C. Stein, J. Wein, *Scheduling Algorithms* , Algorithms and Theory of Computation Handbook, CRC Press, 1997.

[3] M. R. Garey and D. s. Johnson, *Computers and Interactability: A Guide to the theory of NP-completeness*, W. H. Freeman, San Franscisco, 1979.

[4] D. B. West, *Introduction to Graph Theory*, Englewood Cliffs, NJ: Prentice-Hall, 1996.