ABSTRACT

Title of thesis: COMBINATORIAL ALGORITHMS FOR THE ACTIVE TIME AND BUSY TIME PROBLEMS

Saurabh Kumar, Master of Science, 2016

Thesis directed by: Professor Samir Khuller
Department of Computer Science

In this thesis, we consider the problem of scheduling jobs in such a way that we minimize the energy consumption of the machines they are scheduled on. Job scheduling itself has a long and rich history in computer science both from theoretical and applied perspectives. A multitude of different objectives to optimize have been considered such as weighted completion time, penalties for missed deadlines, etc. However, traditional objective functions such as these do not capture or model the energy consumption of the machines these jobs run on. Energy consumption is an important facet of job scheduling to consider not only because of its relationship with the financial costs of scheduling (such as those related to cooling and the cost of powering the machines) but also due to its impact on the environment. This is especially true in the context of data centers as more and more processing is pushed to the cloud. We study two problems related to these issues - the active time problem and the busy time problem. First, we give a purely combinatorial algorithm for the active time problem which matches its best known approximation ratio (the existing algorithm is based on a rather involved LP rounding scheme). Second, we describe

a local search based heuristic for the problem and also consider an experimental evaluation of these algorithms on artificially generated data. Finally, we describe two very simple algorithms which match the current best upper bounds for the busy time problem when all job lengths are equal.

Combinatorial Algorithms for the Active Time and Busy Time Problems

by

Saurabh Kumar

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2016

Advisory Committee:
Professor Samir Khuller, Chair/Advisor
Professor S. Raghavan
Professor William Gasarch

Acknowledgments

I would like to thank my advisor Professor Samir Khuller for all of his guidance over this past year. His insight and knowledge was invaluable during my thesis research. I would also like to thank my thesis committee, Professor William Gasarch and Professor Subramanian Raghavan for their time and helpful feedback. I also appreciate the support of my lab mates Saba, Sheng, Ioana, Ahmed and Manish.

Table of Contents

Chapter 1: Introduction

In this thesis, we consider problems related to energy efficient scheduling of jobs on machines. This is a relatively new area of scheduling research, compared to optimizing traditional scheduling objectives that have a long and rich history. Rather than focusing on the jobs and attempting to optimize some property of the schedule from the point of view of those jobs (such as weighted completion time, minimum penalty, etc.), we focus on the machines and define objectives on them. More specifically, in the context of energy efficiency, we are most concerned with how long a machine is on. A machine is on when it is processing at least one job. We motivate our model (described in the subsequent chapter) with some real world scenarios:

Power Consumption in Data Centers: Consider the following scenario in a data center - we are given a processor which can process g jobs at a given time in parallel, and we are given a memory storage unit (MSU) holding the data that the jobs must access. If at any given time, at least one job is being processed, the MSU must be on (this is independent of the actual number of jobs being processed at the time). It is common for the power being consumed by the processor to be much less than the power being consumed by the MSU (to the extent that we can

assume that the MSU dominates over the processor in terms of energy usage). Our aim is to now minimize the total energy usage of the system i.e minimize the total energy consumption by the MSU.

Efficient Passenger Transportation: Suppose that passengers land at an airport at certain times (known in advance) and are expected to be taken by certain deadlines to their (common) final destination. We have a taxi available to us that can carry at most g passengers at a time. Here, the average time taken by the taxi can be considered to be independent of the number of passengers it is actually carrying. Our aim is to minimize the number of trips the taxi has to make.

Renting Machines in the Cloud: Suppose that we have some jobs with release times and deadlines that need to be processed in the cloud. We can rent a VM with g cores allowing us to process g jobs in parallel. The amount that we pay is dependent only on how long we use the VM and not on the number of jobs we run on it at any given time. Our goal is to schedule jobs on the VM in such a way that we minimize the amount of time we use it.

In all of these examples, the common theme is that the (problem specific) 'cost' of processing jobs at any given time is *independent* of the number of jobs actually being processed at that time. We aim to capture this property in our general model.

This thesis is structured in the following way: In Chapter 2, we will state the active time problem which captures the scenario mentioned previously and describe some existing and new approaches for solving it. First, we describe a greedy algorithm which provides a tight 2 approximation to the problem. This matches the current best approximation for the problem which is based on a rather involved LP

rounding scheme described by Chang et. al. [1]. Our approach is purely combinatorial and leads to a much simpler algorithm and analysis. Next, we sketch a local search heuristic wherein we open at most $b-1$ closed slots and attempt to close at least b while maintaining feasibility. We had initially hoped this could solve the problem optimally; unfortunately, we were able to come up with lower bounds dependent on the parameter b. However, we note that our lower bounds still leave open the possibility that local search could provide a PTAS if the parameter b increases but remains constant. We close out the chapter with an empirical analysis of some of these algorithms.

In Chapter 3, we consider the busy time problem which has been studied previously. The busy time problem differs from the active time problem in that jobs are non-preemptible and we may open up an unlimited number of machines (the objective remains same - we wish to minimize the total amount of time the machines are on). We will describe two very simple algorithms which match the current best algorithms in the case when all jobs have equal length.

Finally, in Chapter 4, we describe some directions for future work.

Chapter 2: The Active Time Problem

2.1 Preliminaries

2.1.1 Problem Definition

The active time model consists of the following: we are given a set of n jobs $J = \{1, 2, ..., n\}$ where each job j has a processing time p_j as well as a valid time interval in which it can be scheduled, defined by a release time r_j and a deadline d_j. Time is divided into unit length slots and we can schedule at most g (distinct) jobs in a single timeslot on the machine. Jobs are pre-emptible (i.e they can be stopped and restarted at integral time points in their intervals). We say that a timeslot is *active* if at least one job is scheduled in it, otherwise we say that it is *closed*. The goal is to find a feasible schedule (one which respects the parallelism parameter g and processing requirements of each job) which minimizes the total number of active slots.

2.1.2 Feasibility of a Schedule

Without loss of generality, we can always assume that a problem instance will have a feasible schedule. This can be checked before running an algorithm using a

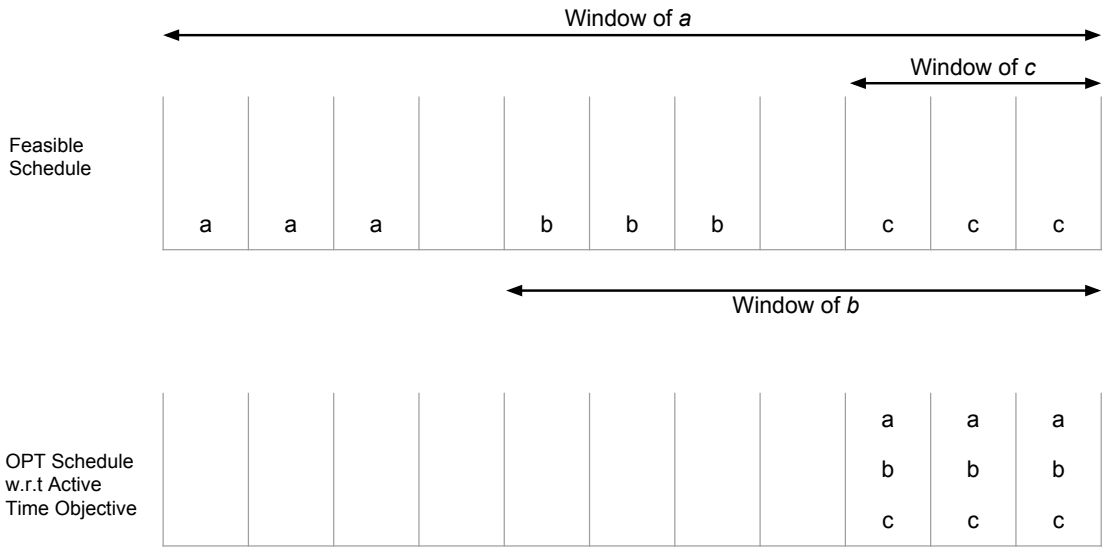

Figure 2.1: The top figure shows a schedule that optimizes a classical objective such as weighted completion time whereas the bottom one shows the optimal schedule with respect to the active time objective. Here, we are most concerned about the number of slots in which the machine is active and not how quickly a job finishes.

max flow computation in the following way: Construct a bipartite graph with the following nodes: (1) A source and sink (2) A node for every job (3) A node for every time slot. Now, we add the following sets of edges: (1) An edge from the source to each job node with capacity equal to the length of the job (2) An edge from each time slot node to the sink with capacity g (3) An edge with unit capacity from each job node to every time slot node in which it is live. The graph is depicted in Figure 2.2. Find the maximum flow possible from the source to the sink. If this flow value is at least the sum of the processing lengths of all the jobs, then a feasible schedule exists for the instance. This schedule is given by the time slot nodes which have flow going through them and the unit capacity edges with flow through them which indicate the assignment of job units to the time slots.

2.1.3 Complexity and Relationship to Network Flow

The key idea here is that the machine consumes a fixed amount of energy per active time slot independent of the number of jobs scheduled in it. Had the cost of a slot been dependent on the number of jobs scheduled in it, the problem could have been solved using min-cost flow - set up a bipartite graph in the way described earlier. Since the cost of scheduling jobs in slot t is linear in the number of jobs scheduled in it (i.e linear in the amount of flow through it), minimizing the total cost of the schedule is equivalent to a min cost flow computation.

As soon as the cost becomes independent of the number of jobs, the problem becomes harder. Intuitively, there is no longer a smooth linear increase of cost with

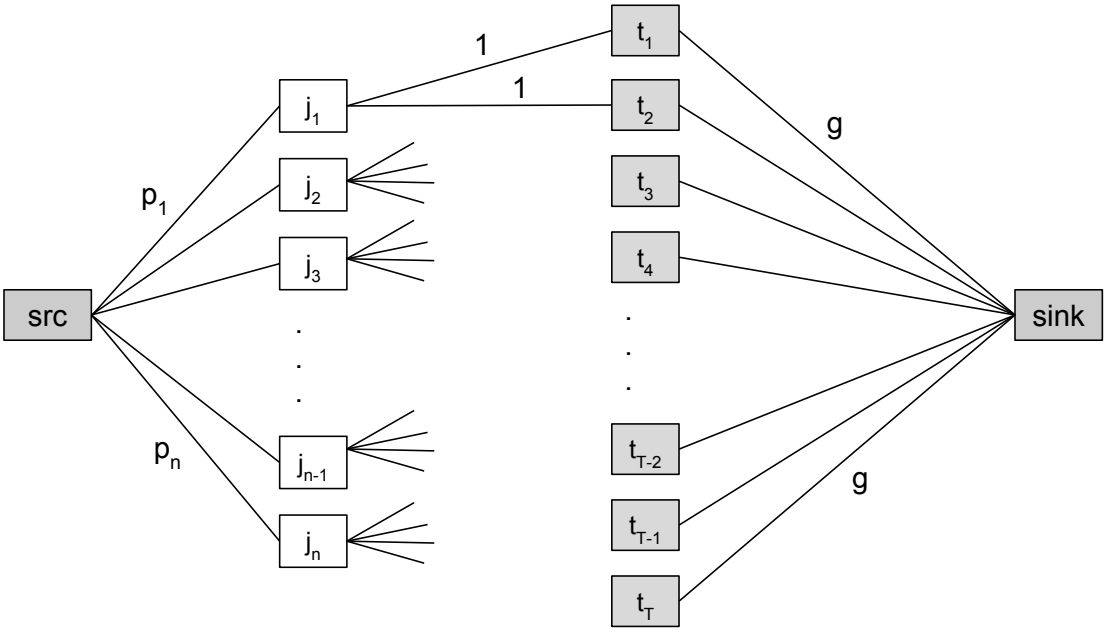

Figure 2.2: A bipartite graph construction used to check whether a given problem instance is feasible.

the number of jobs scheduled in each slot; rather, cost is now a step function which is zero when no jobs are schedued and 1 when at least one is.

If we allow jobs to have multiple intervals in which they can be scheduled (rather than just one which is what we have described in our model), the problem is equivalent to the classical min-edge cost flow problem (in which edge cost is no longer dependent on flow) which is known to be NP hard (listed as problem [ND32] in Garey and Johnson [2]). This problem is also NP hard in the general case (where each job has a set of valid intervals) even when the paralellism parameter g is 3. This can be shown by a reduction from 3 EXACT COVER [3].

However, the complexity status of our model wherein jobs can be scheduled in single contiguous intervals is as yet unknown.

2.2 Related Work

In this section, we briefly look at exact algorithms for a special case of the active time problem. Then we move to the two existing approximation algorithms for the active time problem - the minimal feasible solution (MINFEAS) and the LP rounding approach.

2.2.1 Exact Algorithms for Some Special Cases

In the case when we allow arbitrary parallelism but restrict ourselves to unit length jobs, the active time problem has exact solutions. Chang, Khuller and Gabow solve this problem exactly using the Lazy Activation Algorithm (LAZYACT) [3].

LAZYACT first scans the jobs and modifies their deadlines so that for any time slot, at most g jobs have their deadlines at that time. Then, the algorithm opens slots based on an earliest deadline first strategy and schedules jobs accordingly.

This special case is also equivalent to the one dimensional capacitated rectangle stabbing problem. In this problem, we are given a collection of 1D rectangles (i.e intervals on a line) and a set of vertical lines 'stabbing' (intersecting) them. Each vertical line has a cost and a capacity which indicates how many rectangles it can intersect. The problem asks us to find the set of lines of minimum cost that hit all the rectangles at least once. In our scenario, the rectangles represent the job windows and the vertical lines are the timeslots each with capacity g and unit cost. This interpretation leads to a dynamic programming algorithm described by Even et. al. [4] and based on ideas introduced by Baptiste [5] for the minimum gap scheduling problem.

We note that the case when we allow arbitrary job lengths but restrict ourselves to a unit parallelism parameter is trivial - the amount of time taken is simply the sum of the job lengths.

2.2.2 Minimal Feasible Solution

Interestingly, it turns out that the minimal feasible solution (a solution where if we close any timeslot, the schedule becomes infeasible) is already a 3 approximation to the optimal solution. Further, this ratio is tight. This algorithm (MINFEAS) and its analysis was described by Chang, Mukherjee and Khuller in [1] as well as in

Koyel Mukherjee's PhD thesis [6].

Algorithm: Given any feasible schedule (obtained using a max flow computation), we randomly attempt to close slots currently open. On closing a slot, we check whether a feasible schedule can still be obtained from the slots still open. If so, we keep that slot closed, otherwise we open it. We repeat this procedure until no slot can be closed without making the schedule infeasible.

All of these operations are performed on the bipartite graph shown in Figure 2.2. Obtaining a feasible initial schedule can be done using a max flow computation as described earlier. Closing a time slot is equivalent to setting the capacity of the edge joining that time slot node to the sink to zero. Opening a time slot is equivalent to setting the capacity of the corresponding time slot node to sink edge to g.

Analysis Sketch: Denote the minimum feasible solution obtained by MINFEAS by S and the optimal solution by OPT. Let the set of non-full slots in S be S_N and the set of full slots be S_F. Now, in every slot in S_N, there must exist at least one job that cannot be moved into another slot (for otherwise, we could have closed this slot by moving every job out). This could only have happened because this job was scheduled in every non-full slot in its window. We will say that such a job is *non-full rigid* (NFR) and call the set of such jobs J_{NFR}.

We will charge the cost of non full slots to J_{NFR}. It is proved in [1, 6] that from J_{NFR} we can identify a subset of NFR jobs say J^*_{NFR} such that the following two conditions are satisfied: (1) every slot in S_N contains at least one job from J^*_{NFR} and (2) at most two of the jobs in J^*_{NFR} are live at any given slot. Intuitively, the set J^*_{NFR} consists of jobs whose windows span over the non full slots in S contiguously

Figure 2.3: Example of the J^*_{NFR} job set. The boxes represent job windows. The light shaded jobs (set J_1) represent one partition of jobs with disjoint windows and the dark shaded jobs (set J_2) represent the other such partition. Every non full slot in S is covered by this set.

with the maximum degree of overlap being 2. This is shown in Figure 2.3.

Firstly, from property (1), we see that $|S_N| \leq \sum_{j \in J^*_{NFR}} p_j$. Secondly, we see that J^*_{NFR} can be split into two sets of jobs (J_1 and J_2) such that all the jobs within the set have disjoint windows (we will also refer to these as tracks). The sum of processing times of jobs in each track is a lower bound on the size of OPT (due to disjoint windows) i.e $\sum_{j \in J_1} p_j \leq \text{OPT}$ and $\sum_{j \in J_2} p_j \leq \text{OPT}$. Therefore, since $J^*_{NFR} = J_1 \cup J_2$, $|S_N| \leq \sum_{j \in J^* NFR} p_j \leq \sum_{j \in J_1} p_j + \sum_{j \in J_2} p_j \leq 2\text{OPT}$.

Further, the full slots can be charged directly to the optimal solution (clearly, the number of full slots in S cannot exceed the size of the optimal solution) i.e $|S_F| \leq$ OPT. Finally, combining the previous two results, we get $|S| = |S_N| + |S_F| \leq 3\text{OPT}$.

2.2.3 LP Rounding Approach

There also exists a rounding approach of the natural LP relaxation of the active time problem as described in [1, 6]. The rounding scheme is rather involved but achieves a 2 approximation for the problem (it is shown that the integrality gap

of the LP is also 2).

2.3 Greedy Algorithm

In the greedy algorithm (GREEDY), we scan slots from left to right and shut them down if feasible (feasibility is again checked using max flow). We can interpret this as a form of local search where we attempt to close down a particular slot t and open every slot $t' > t$ and see if we can ensure feasibility. If so, we keep t closed, otherwise we open it and continue.

We will prove the following approximation ratio for the greedy algorithm:

Theorem 1. *GREEDY is a 2 approximation to the active time problem.*

The remainder of this section is devoted to the proof of Theorem 1. In order to prove this, we will introduce a framework of arcs to compare our solution to OPT.

2.3.1 Arcs Framework

Initially, we will assume that the schedule we are working with is the output of an arbitrary algorithm ALG (since the properties we describe are independent of the algorithm we choose to obtain the schedule). The only assumption we make is that the schedule ALG outputs is minimal and feasible. We will later consider the special case when ALG is GREEDY in order to prove Theorem 1.

This framework is used to compare the solution of ALG to OPT. Suppose we have obtained a solution using ALG that is minimal and feasible. We let S^* denote the optimal schedule and S denote the schedule obtained by ALG. Let the

set of jobs scheduled by OPT and ALG in time slot t be denoted by $J^*(t)$ and $J(t)$ respectively.

We place both schedules S and S^* on the same timeline. Interpreting each slot as a vertex in a graph, we will draw arcs from slots opened by S to slots opened by S^*. These arcs correspond to the movement of jobs from their positions as assigned by ALG to their correct and optimal positions as assigned by OPT. Intuitively, if we construct an arc from slot t in S to slot t^* in S^*, then that means that some set of jobs that was scheduled by ALG in t was scheduled by OPT in t^* and the arc now represents the 'movement' of those jobs to their correct position.

We construct the arcs in the following way:

1. Scan OPT slots from right to left. For each such OPT slot t^*,

   (a) Consider the jobs from the set $J^*(t^*)$ that are scheduled by ALG in S after t^*. Shift these jobs to the left (towards t^*) along slots in S as much as possible.

   (b) Now scan the slots in S at or after t^* from left to right. For each such ALG slot t considered, check if t contains one or more jobs in common with t^* that haven't been charged to any other slot in OPT. If there are any, add a left arc from t to t^*. We now say that those jobs are *charged* to t^* (note that we charge as many jobs in common as we can from t to t^*).

2. Draw right arcs from ALG slots with uncharged jobs arbitrarily to the slots in OPT which also have those jobs uncharged. Once again, we shift the jobs

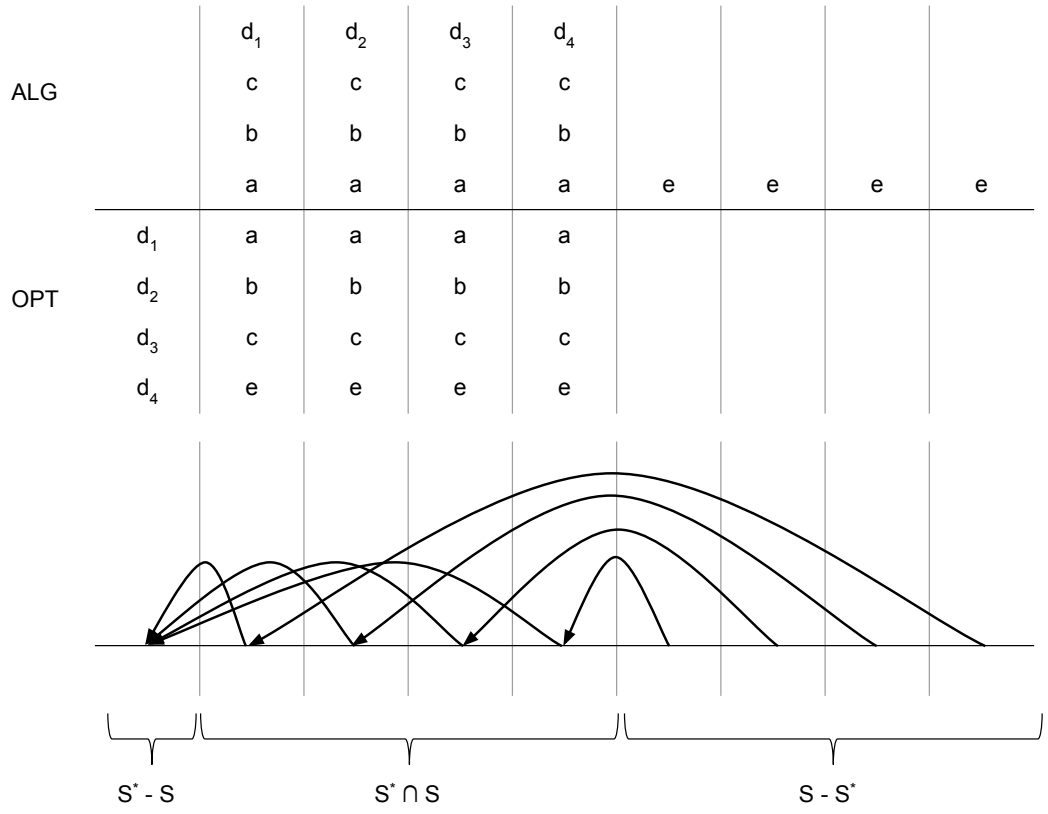

Figure 2.4: An example of arcs constructed based on the schedule (the letters represent job units). Here, $g = 4$.

along the direction of their arcs as much as possible.

Note that here, we have implicitly assumed that due to minimal feasibility, no slot can become empty as a result of this shifting. An example of such a construction of arcs is shown in Figure 2.4.

The ALG schedule S has two types of slots: $S - S^*$ slots and $S \cap S^*$ slots. We now bound the number of non-full slots in $S - S^*$ using the notion of a *primary*.

Definition 1. The *right primary* of S slot t is defined to be the earliest S^* slot

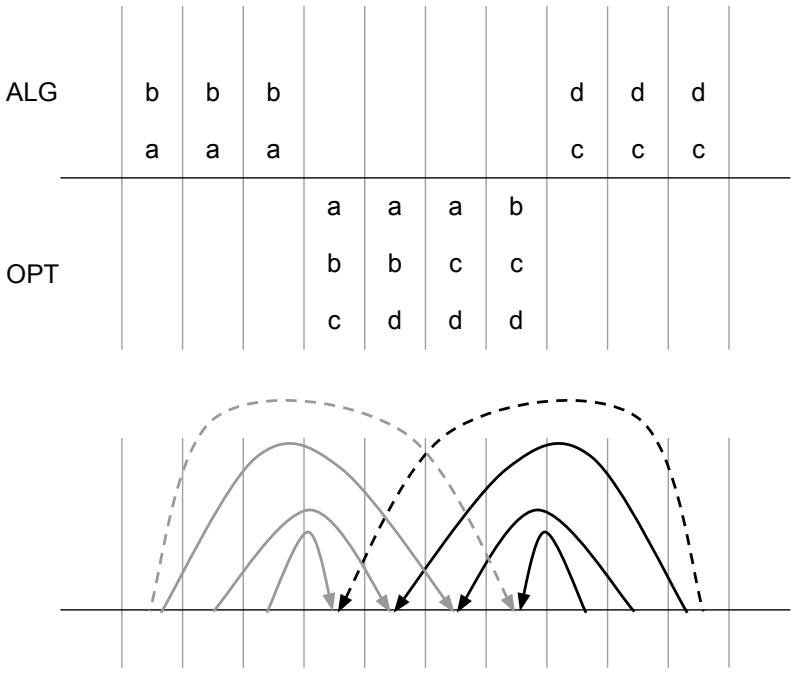

Figure 2.5: An example of primaries for the given schedule. The solid gray and black lines are the right and left primaries respectively, the dashed gray and black lines are non-primary arcs

after t that it charges. The *left primary* of S slot t is defined to be the latest S^* slot before t that it charges.

Intuitively, an S slot t only charges slots later than its right primary and earlier than its left primary. We will say that an S slot t *left charges* (or is *left-charging*) if it has a left primary and it *right charges* (or is *right-charging*) if it has a right primary. An example of primaries is shown in Figure 2.5.

The next lemma will be used to define our charging scheme for non-full $S - S^*$ slots.

Lemma 1. *No two non-full $S - S^*$ slots share the same left primary.*

Proof. Consider two non-full left charging $S - S^*$ slots t_1 and t_2 with $t_1 < t_2$ as shown in Figure 2.6. If possible, let t^* be the left primary for both t_1 and t_2. Let J_1 and J_2 be the jobs charged by t_1 and t_2 to t^* respectively.

After the first right to left scan, only left charging arcs have been constructed. Then, since we shift in the direction of the arcs, the jobs charged to t^* by t_2 must form a subset of $J(t_1)$ since otherwise they would have been left shifted to t_1 since t_1 is non-full. Here, we take advantage of the fact that job intervals are single and contiguous - if the jobs in J_2 were live at t_2 and t^* but not at t_1, we could not have guaranteed the previous statement. Therefore any job being charged to t^* by t_2 could also have been charged by t_1. But by our arc construction method, this is what should have happened. Therefore, t_2 must have as its left primary, a slot earlier than t^*. □

Here, we point out that a left primary of a slot may be charged by other left charging arcs as well, just not left primaries. An example is shown in Figure 2.7.

Now, in the next two lemmas, we will characterize the structure of the $S \cap S^*$ slots.

Lemma 2. *In any slot in $S \cap S^*$, the total (possibly zero) incoming job mass (from incoming arcs) must equal the total outgoing job mass (from outgoing arcs).*

Proof. Based on the shifting we perform in our arc construction, we are moving the jobs in the direction of their arcs as much as possible. In an $S \cap S^*$ slot t, we argue that the total incoming job mass $J_{in}(t)$ must equal the total outgoing job

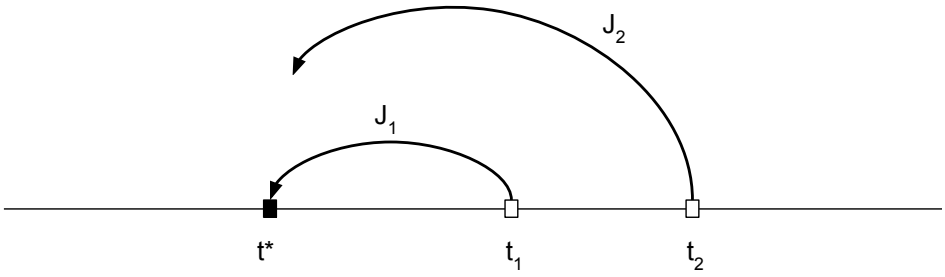

Figure 2.6: Here, the white boxes denote S slots while the black one denotes an S^* slot. The left primaries for both t_1 and t_2 cannot be t^* since here $J_2 \subseteq J(t_1)$ due to left shifting. So here, the left primary from t_2 as shown cannot terminate in t^*.

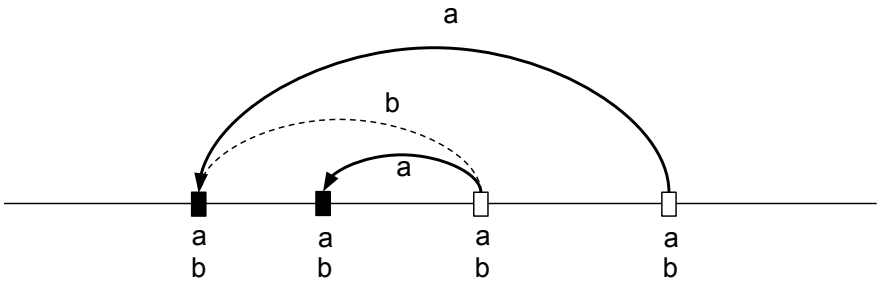

Figure 2.7: In this example, the black boxes denote OPT slots whereas the white ones denote S slots. The left primaries are shown in solid black arcs whereas the dashed line is a left charging arc (not a primary) that charges the left primary of a slot.

mass $J_{out}(t)$. Consider the possible cases: (1) if $J_{in}(t) > J_{out}(t)$, then t must have had at least $J_{in}(t) - J_{out}(t)$ amount of free space. Our arc construction algorithm due to the shifts, would have then moved $J_{in}(t) - J_{out}(t)$ worth of job mass (the net incoming excess) into t in our ALG schedule S. (2) if $J_{in}(t) < J_{out}(t)$, then we are creating free space of size $J_{out}(t) - J_{in}(t)$ in t in S. This is unnecessary, and we can update OPT by moving the outgoing $J_{out}(t) - J_{in}(t)$ size job mass into t in the OPT schedule so that there is no longer any excess job mass moving out. □

Lemma 3. *All slots in $S \cap S^*$ which have incoming arcs (and therefore outgoing arcs) are full. All non-full slots in $S \cap S^*$ have no incoming or outgoing arcs.*

Proof. From Lemma 2, for any $S \cap S^*$ slot t with an incoming arc, the total incoming job mass equals the outgoing job mass. The only reason we cannot move any more of the incoming job mass into t must be because t is full.

Consider any non-full $S \cap S^*$ slot t'. If it had any incoming arcs, we should have been able to move some of that job mass into t' (since it is non-full). Since this does not happen, this can only mean that t' has no incoming arcs (and therefore no outgoing arcs either from Lemma 2). □

2.3.2 Analysis of GREEDY

We will use the arcs framework introduced in the previous section to prove the approximation ratio for GREEDY. Suppose we create the arcs using the method described earlier. We denote the GREEDY schedule by S_G and the optimal schedule by S^*. We will need one more property of our arc construction method for this

analysis:

Lemma 4. *If any non-full $S_G - S^*$ slot t is crossed by a left arc, t too must left charge.*

Proof. Suppose the crossing arc comes from S slot t' (and therefore $t' > t$). Due to left shifting, the contents being charged by the crossing arc must also be present in t (since t is non-full). Suppose the arc terminates in the S^* slot t^*. When we construct arcs in our right to left pass, for the given S^* slot t^*, we consider ALG slots from left to right after t^* when adding arcs. Therefore, when considering jobs in S slots to charge to t^*, we would have considered t before t' and added an arc (unless possibly t was already left charging some other slot after t^*). Either way, t must left charge an S^* slot.

□

The following lemma characterizes our greedy solution:

Lemma 5. *Every non-full $S_G - S^*$ slot must left charge.*

Proof. Suppose there exists a $S_G - S^*$ slot t that only right charges. Then, from Lemma 4, there is no left arc that crosses t. This means that OPT was able to schedule all the jobs in t later than t, without using any slots earlier than t (since there is no left crossing arc over t). But in this scenario, GREEDY would have then closed t and opened possibly every slot later than t (which would subsume the set of slots in S^* opened after t) to schedule its jobs. Therefore, such a slot t is not possible. □

We can now prove Theorem 1:

Proof. Based on Lemma 3, we consider two disjoint subsets of the OPT schedule: S^*_p and S^*_{np}. The slots in S^*_p are the left or right primary of at least one slot in S and $S^*_{np} = S^* - S^*_p$. Note that here, the slots in S^*_{np} consist of two types - those that have incoming (and outgoing) arcs but none of these arcs are primaries, and those that have no incoming or outgoing arcs at all. Clearly, we have that $|S^*_p| + |S^*_{np}| = |S^*|$.

We will bound the following three classes of slots in S_G separately:

Non-full slots in $\mathbf{S_G - S^*}$: Since each non-full $S_G - S^*$ slot left charges (by Lemma 5), it must have a left primary. Combining this with Lemma 1, we see that the number of non-full $S_G - S^*$ slots in GREEDY cannot exceed $|S^*_p|$.

Non-full slots in $\mathbf{S_G \cap S^*}$: From Lemma 3 and the definition of S^*_{np}, the set of non-full $S_G \cap S^*$ slots is completely contained in S^*_{np}. We directly charge these slots to S^*_{np}.

Full slots in $\mathbf{S_G}$: The full slots in S_G are directly charged to OPT. Clearly, by the mass bound, the number of full slots in S_G cannot exceed OPT.

Putting everything together, the total cost of our GREEDY solution is upper bounded by $|S^*_p| + |S^*_{np}| + \text{OPT} \leq 2\text{OPT}$. □

Tight Example for GREEDY: Suppose the set of jobs consists of g jobs of unit length with window $[1, g+2)$, $g-1$ rigid jobs with window $[2, g+2)$ and one job of length g with window $[2, 2g+2)$. Then, OPT would have opened the $[1, 2)$ slot, scheduled all unit jobs there and then been able to schedule the long job above the set of rigid jobs. However, our algorithm, will greedily close the $[1, 2)$ slot thereby

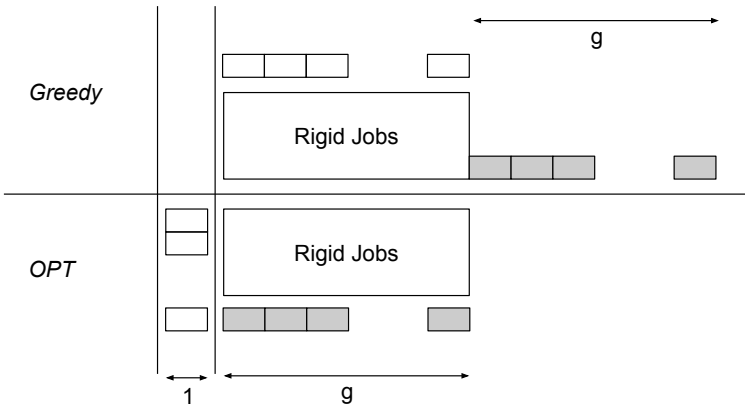

Figure 2.8: Lower Bound of 2 for GREEDY. The rigid jobs are shown as a box. The unit length jobs are shown as the small white rectangles. The long g length job is shown composed of multiple job units in the form of shaded rectangles.

forcing the unit jobs to be scheduled above the rigid jobs and pushing the long job out. Figure 2.8 shows the example. The top half shows the schedule obtained by GREEDY (with a cost of $2g$) while the bottom half shows the optimal one (with a cost of $g + 1$).

2.3.3 Alternate Analysis for MINFEAS

Using our arcs framework, we can derive an alternate proof that the minimal feasible solution is a 3 approximation to OPT. Let S_M be the schedule obtained by MINFEAS and let S^* be the optimal schedule. We divide S^* into S^*_p and S^*_{np} as per their definitions.

In its current form, Lemma 1 cannot be extended to right primaries due to the arbitrary way in which we construct the right arcs (an example of two slots having

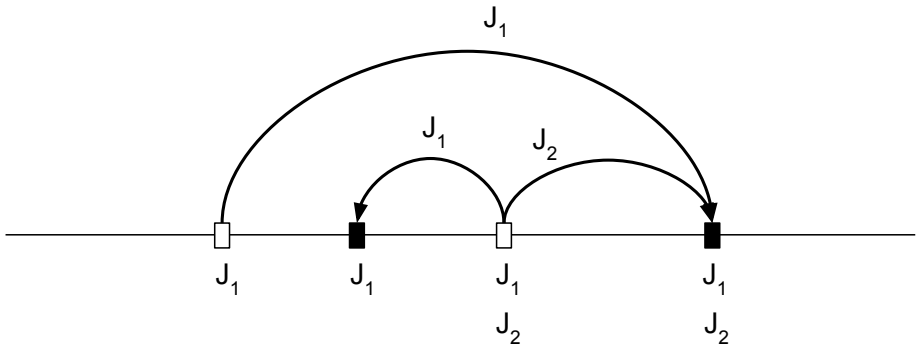

Figure 2.9: The right primaries of both t_1 and t_2 are the same. The arcs have been constructed according to the procedure described previously. J_1 and J_2 denote two arbitrary sets of jobs assigned/charged to the slots/arcs which have them as labels. The white boxes are S slots whereas the black ones are S^*.

the same right primary is shown in Figure 2.9).

We will define a procedure to rearrange the right charging arcs so that we can guarantee uniqueness of right primaries as well.

Right Charging Arcs Rearrangement: Consider the following general scenario depicted in Figure 2.10. Suppose t_N and t (with $t < t_N$) are two non-full $S - S^*$ slots that have the same right primary t^* such that t_N is the latest S slot earlier than t^* which charges it. Then, because of right shifting, $J \subseteq J(t_N)$. Therefore, t_N must charge at least one other S^* slot to charge the job set J which it contains and cannot charge to t^*. We show the three general possibilities as $J_{[t,t_N]}$, $J_{>t^*}$ and $J_{<t}$ in the figure. $J_{[t,t_N]}$ is charged in the range $[t, t_N]$, $J_{>t^*}$ is charged after t^* and $J_{<t}$ is charged before t (note that t_N cannot charge any slots in the range $[t_N, t^*]$ since t^* is its right primary).

First of all, we argue that $J \cap J_{<t} = \emptyset$. Assuming that $J_{<t}$ is non-empty, due to Lemma 4, t must also left charge. Suppose if possible, there was a job $j \in J$ such that $j \in J_{<t}$. Then during our first right to left scan, we would have been able to charge it to a slot to the left of t. Furthermore, in the remainder of this rearrangement procedure, we will only (potentially) remove left arcs from slots nearest to their right primaries and once we do so, we never process those slots again; so t has not had any left arcs removed (because if it had, we would not have been currently processing it). Therefore, such a j cannot belong to J either because it is charged by t to its left or because such a j does not exist. Either way, $J \cap J_{<t} = \emptyset$.

Then we must have that $J \subseteq J_{[t,t_N]} \cup J_{>t^*}$. Our rearrangement procedure will work in the following way - we will take all of the job mass from $J_{[t,t_N]}$ and $J_{>t^*}$ in common with J (that t_N charges to slots in the range $[t, t_N]$ and later than t^*) and have t_N charge it to t^*. Therefore, we can now remove the right primary arc of t since all of the job mass it was charging to t^* has now already been charged by t_N. But this surplus job mass in set J from t still needs to be charged somewhere. This we will redistribute to all of the S^* slots that t_N was charging and we took the jobs from.

More formally, if t_N charged an S^* slot t_1^* with job set J_1, then we remove jobs $J_1 \cap J$ from that arc and have t_N charge them to t^* via its right primary. We will then add an arc from t to t_1^* with the same $J_1 \cap J$ job set (to account for the jobs we just removed). We will continue this till all of J from t has been right charged to S^* slots other than t^*.

We repeat this for all such $t < t_N$ for whom t^* is a right primary. Ultimately,

only the slot nearest t^* on the left and charging it, will have t^* as a right primary. Further, because we are only adding right arcs and removing left arcs (including possibly left primaries) in the worst case, we do not affect the correctness of Lemma 1.

If this created new situations where the right primaries coincided, then we reapply the above procedure. Note that repeated applications of this procedure will not affect slots already processed because the right primary of an S^* slot (to which this procedure has been applied) is fixed to be the nearest S slot earlier than it and charging it. So once a slot t^* is processed by the procedure, no right charging primary arcs are added to it because any job that could have charged it from the right would already have been contained in t_N (its arc would cross t_N since t_N is the slot nearest to t^* to its left and charging it) and therefore t_N itself would have contributed any jobs it carried. Thus, this process will terminate and no two non-full $S - S^*$ slots will have the same right primary.

The above argument allows us to state the following lemma:

Lemma 6. *After applying the right charging arcs rearrangement procedure, no two non-full $S - S^*$ slots have the same right primary.*

Now we proceed with the proof of the 3 approximation to MINFEAS. We will first bound the non-full slots in $S_M - S^*$. Consider, from Lemmas 1 and 6, the number of non-full $S_M - S^*$ left charging slots and non-full $S_M - S^*$ right charging slots are both upper bounded by $|S_p^*|$. Therefore, the total number of non-full $S_M - S^*$ slots is upper bounded by $2|S_p^*|$. Now, similar to the analysis

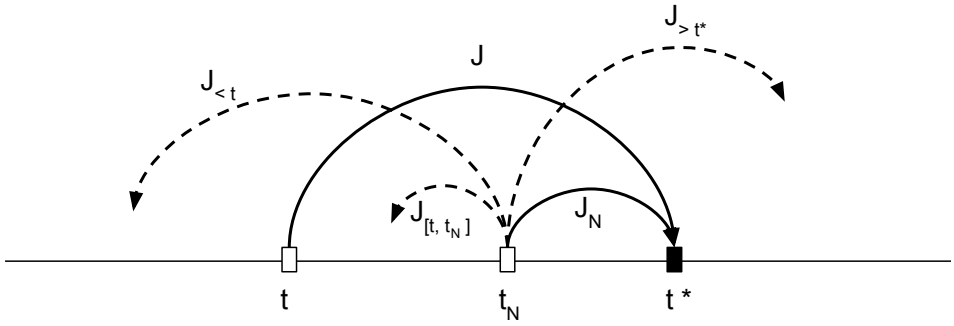

Figure 2.10: White boxes denote S slots while the black box is in S^*. t and t_N share the same right primary t^*.

of GREEDY, the full slots in S_M are charged to OPT and the non-full slots in $S_M \cap S^*$ are charged to S^*_{np}. Thus, the cost of MINFEAS is upper bounded by $2|S^*_p| + \text{OPT} + |S^*_{np}| \leq 3\text{OPT}$.

2.4 Local Search

Intuitively, in our previous algorithms, a reason our solution is off from the optimal by such a large factor is due to the fact that we only close down slots, we never open them. Therefore, if we happened to close down a time slot early on that may have been useful later, we cannot open it again. We pay the penalty of incorrectly closing that slot via an increased approximation ratio. Therefore, as a way around this problem, we consider a local search strategy. Our local operation is defined in terms of a parameter b as follows: we will attempt to close at least b slots that are currently open and open up at most $b-1$ currently closed slots

in order to maintain feasibility. We will repeatedly apply this operation to our minimal feasible schedule until it is no longer possible. Note that in this local operation, we will always close at least one slot more than we open and therefore make progress. The parameter b is a constant upon which the running time of this operation depends exponentially. We will denote the local search algorithm with parameter b as LOCAL(b).

For LOCAL(b), the best lower bound we have is $\frac{b}{b-1}$. We will describe the example in terms of job mass - suppose we have OPT scheduled as a single mass of jobs of length OPT and height g. Suppose LOCAL(b) schedules this job mass on the left and right sides of this OPT mass as shown in Figure 2.11. Now, the cost of the LOCAL(b) solution is the number of open slots. We will assume that the job mass scheduled by LOCAL(b) on the left is live in the OPT region but not in the region to the right (and vice versa for the right region). We require that for any b slots we wish to close, the total job mass in them be at least $(b-1)g+1$ thereby forcing us to open at least b slots. This would mean that no local operation is possible. Assuming symmetry, we require that each slot schedules at least $\frac{(b-1)g+1}{b}$ job mass in it. By conservation of job mass, we must have that ALG $\times \frac{(b-1)g+1}{b} =$ OPT $\times g$. Taking g to the limit of infinity, we obtain the aforementioned lower bound.

For completeness, we also consider the case when we apply our local search heuristic not simply to a minimal feasible solution, but to the solution obtained by GREEDY. In that case, we can show a lower bound of $\frac{2b}{2b-1}$ in the following way: Suppose we have $x \times g$ unit jobs (we will define x later) with window $[0, x+g]$, $g-x$ rigid jobs of length g with window $[x, x+g]$ and x jobs of length g with

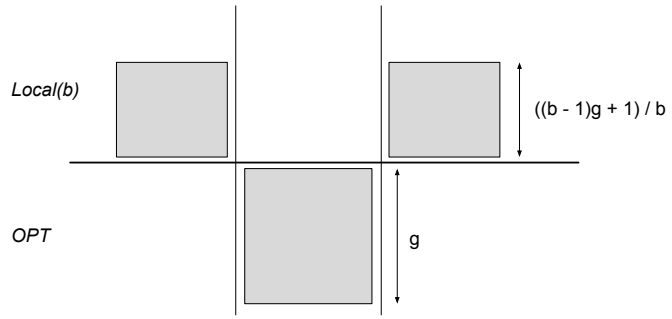

Figure 2.11: Lower Bound of $\frac{b}{b-1}$ for LOCAL(b). The shaded boxes represent job mass.

window $[0, x + 2g]$. Ideally, it would generally make sense to open the slots in the range $[0, x]$ so as to schedule the unit jobs there and allow the g length jobs to be scheduled above the rigid ones. This is shown in Figure 2.12. Now, considering the LOCAL(b) heuristic, if we opened $b - 1$ of the x slots at the beginning, we would be able to schedule $(b - 1)g$ units of jobs there. This in turn would imply that we could free up $\frac{(b-1)g}{x}$ slots over the rigid jobs. Then we could move the long jobs in partially over the rigid ones, allowing us to close $\frac{(b-1)g}{x}$ of the right most slots. Because our solution is local optimal, we must then have that $\frac{(b-1)g}{x} < b$ thus giving $x > \frac{(b-1)g}{b}$. Therefore, the local optimal solution has cost $2g$ whereas OPT has cost $x + g > \frac{(b-1)g}{b} + g$. The aforementioned lower bound follows.

2.5 Empirical Evaluation

We studied the practical utility of MINFEAS and GREEDY by implementing them and testing them on artificially generated data.

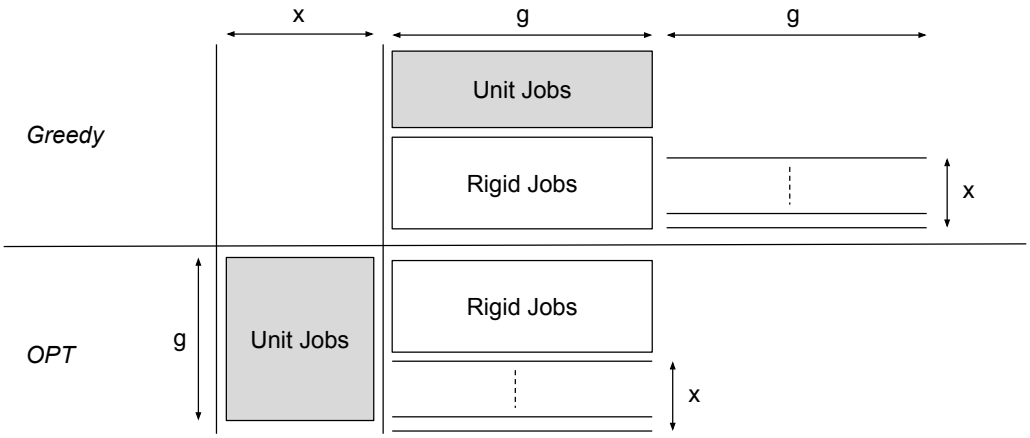

Figure 2.12: The $\frac{2b}{2b-1}$ lower bound for LOCAL(b) applied to the output of GREEDY.

2.5.1 Implementation Details

Our implementations of MINFEAS, GREEDY were done in C++. Because MINFEAS and GREEDY involve repeated max flow computations (to perform feasibility checks after each slot is closed), we used the EIBFS algorithm [7, 8] which works well for this use case. The implementation of EIBFS was obtained from the Maximum Flow Project [9] To obtain optimal solutions to compare with, we solved the LP of the active time problem using Gurobi 6.5.2 [10] in Python. All tests were run on a Dell Inspiron laptop with 8GB RAM and an Intel i7 processor with a clock frequency of 2.50GHz.

2.5.2 Artificial Data Generation

We used an approach for generating data similar to the one described in Jessica Chang's PhD thesis [11], namely, we defined a problem instance using a tuple of 5 parameters (N_1, N_2, M_1, M_2, B). Here, the upper bound on the number of jobs in the schedule n was selected randomly from the range $[N_1, N_2]$, the total number of time slots T was selected randomly from $[M_1, M_2]$ and the parallelism parameter g was selected randomly in the range $[\frac{B}{2}, B]$ (note that in [11], the author selects the parallelism parameter randomly in the range $[1, B]$). We generated jobs in two ways - random and adversarial. In a given schedule, we chose to generate a random job or an adversarial unit with some probability.

Random Jobs: For a random job j, we generated a random release time $r_j \in [0, T]$, a random deadline $d_j \in [r, T]$ and a random processing time $p_j \in [1, d_j - r_j]$. This job was added to the list of jobs already generated and the instance was checked for feasibility (using a max flow computation). If feasible, we continued the process. However, if the job added made the schedule infeasible, we dropped it and generated another random job.

Adversarial Jobs: To generate adversarial jobs, we constructed the following 'adversarial' unit (note that this is similar to the greedy lower bound construction) as depicted in Figure 2.13. The unit starting at position t would consist of $x \times g$ unit jobs with window $[t, t + x + g]$, $g - x$ rigid jobs of length g with window $[t + x, t + x + g]$ and x jobs of length g with window $[t, t + x + 2 \times g]$.

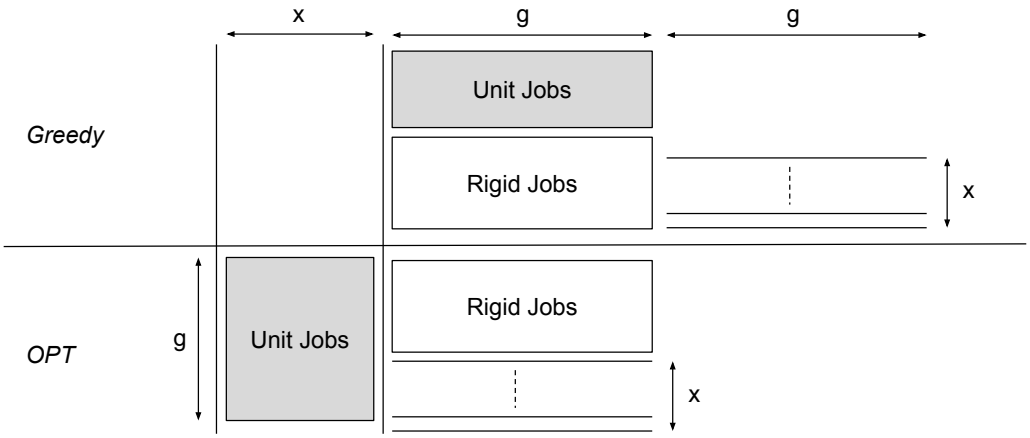

Figure 2.13: An adversarial unit - the upper half indicates how a greedy algorithm might schedule the jobs whereas the lower half shows the optimal schedule for the unit.

Ideally, it would generally make sense to open the slots in the range $[t, t+x]$ so as to schedule the unit jobs there and allow the g length jobs to be scheduled above the rigid ones. If however, any of our algorithms close one or more of the slots in that range, those unit jobs will be forced to be scheduled over the rigid jobs thereby driving the g length jobs out increasing the number of open slots. This unit was added to the list of jobs and checked for feasibility. If feasible, we continued the process, otherwise we deleted the unit.

If 100 consecutive random jobs or adversarial units generated were all infeasible, we aborted the procedure for the particular instance and used whatever jobs we had already added.

We generated two testbeds each consisting of 100 problem instances -

	Random	Mixed
Number of Problem Instances (PI)	100	100
(N_1, N_2, M_1, M_2, g)	25, 50, 0, 200, 10	25, 50, 0, 200, 10
Average Number of Jobs per PI	35.6	46.3
Std Dev of Jobs per PI	8.4	11.5
Average Number of Time Slots per PI	107.2	83.9
Std Dev of Time Slots per PI	54.6	49.0
Adversarial Unit Probability	0	0.5

Table 2.1: Details of the two testbeds generated.

Random: A problem instance was generated using the technique described above and consisted only of randomly generated jobs.

Mixed: A problem instance was characterized by the tuple (described above) and in the process of generating a problem instance, on each iteration, we decided to add a random job or an adversarial unit with equal probability.

The details of these testbeds are shown in Table 2.1.

2.5.3 Solution Quality

In general, both algorithms fared better on Random as opposed to Mixed as was expected. However, the generation of adversarial units affected GREEDY far more than MINFEAS. The behavior of MINFEAS and GREEDY on these two testbeds is shown in Tables 2.2 and 2.3. It is possible that the randomness of MINFEAS allows it to overcome the adversarial nature of the input better than

	MINFEAS	GREEDY
Number of optimal solutions (out of 100)	58	97
Mean ALG/OPT Ratio	1.009	1.0005
Max ALG/OPT Ratio	1.1	1.03

Table 2.2: Statistics of MINFEAS and GREEDY on the Random testbed.

	MINFEAS	GREEDY
Number of optimal solutions (out of 100)	35	15
Mean ALG/OPT Ratio	1.08	1.22
Max ALG/OPT Ratio	1.33	1.71

Table 2.3: Statistics of MINFEAS and GREEDY on the Mixed testbed.

GREEDY.

2.5.4 Time Requirements

Time required depended on the number of time slots to consider, with the time taken to run MINFEAS and GREEDY both generally increasing with increasing number of time slots. There did not seem to be as clear a relationship with the number of jobs. The biggest bottleneck was the repeated max flow computations though the nature of the EIBFS algorithm mitigated this problem somewhat (by reusing values from prior flow computations to compute new flows). The timing statistics of MINFEAS and GREEDY on the Random and Mixed testbeds are shown in Tables 2.4 and 2.5 respectively. We believe that the times taken for the Mixed

	MINFEAS	GREEDY
Average Time Taken (s)	0.005	0.002
Maximum Time Taken (s)	0.040	0.018

Table 2.4: Timing statistics of MINFEAS and GREEDY on the Random testbed.

	MINFEAS	GREEDY
Average Time Taken (s)	0.0008	0.0003
Maximum Time Taken (s)	0.0036	0.0013

Table 2.5: Timing statistics of MINFEAS and GREEDY on the Mixed testbed.

testbed are smaller than the Random testbed possibly due to the fact that the average number of timeslots for each problem instance in the mixed testbed was smaller.

Chapter 3: The Busy Time Problem

3.1 Preliminaries

3.1.1 Problem Definition

In the busy time problem, we are given a set of n jobs $\mathbb{J} = \{1, 2, ..., n\}$ where each job j has a release time r_j, deadline d_j and processing time p_j. Unlike the active time model, jobs here are not preemptible. We are given an unbounded set of machines and each can schedule at most g distinct jobs at any given time. A machine is said to be busy at a time t if it is processing at least one job at t, otherwise it is said to be idle. The busy time of any feasible schedule is the sum of the busy times of each machine opened. Our goal is to schedule all the jobs on *any* number of machines in a feasible way (the parallelism parameter g and the processing requirements of each job are respected) such that the total busy time of the schedule is minimized.

3.1.2 Related Work

The busy time problem has been considered both in the general form described above as well as a more restricted form where each job's window is equal to the length

of its processing time. Such a job whose processing time is equal to the length of its window is called an interval job. The busy time problem both with interval jobs and without is NP hard. This was proved in the context of the fiber minimization problem by Winkler and Zhang [12].

Flammini et. al. [13] introduced the busy time problem in its current form and presented a simple algorithm called FirstFit that achieved a 4 approximation for interval jobs. They also presented a $2+\epsilon$ approximation (Khandekar et. al. [14] showed that a small change could improve it to a $1+\epsilon$ approximation) for bounded length jobs and a 2 approximation when the input formed a proper interval graph.

Khandekar et. al. [14] consider a more general problem with non-interval jobs that not only have release times, deadlines and processing times, but also a resource demand. In other words, rather than allowing at most g jobs to be scheduled in parallel on a machine, they considered a more general scenario where the resource demands of all the jobs scheduled on a machine could not exceed g. For this problem, they gave a 5 approximation. They also gave improved results for some special cases of inputs as well. More specifically, they gave a dynamic programming algorithm to solve the busy time problem exactly when unbounded parallelism was allowed (i.e $g = \infty$). Chang et. al. [1] used the DP to obtain a 3 approximation for the busy time problem (with non-interval jobs) using the GreedyTracking algorithm. Their algorithm worked by selecting 'bundles' of tracks (a set of jobs with disjoint spans) from the set of unscheduled jobs and scheduling each bundle on a separate machine.

3.2 Approximation Algorithms

In this section, we give 4 and 3 approximations for the busy time problem when jobs have equal lengths. We use the following notation: Let OPT denote the optimal solution to the busy time problem for finite parallelism parameter g. Let OPT_∞ denote the optimal solution obtained when we allow unbounded parallelism (i.e $g = \infty$). The case with unbounded parallelism is polynomial time solvable using a dynamic program as described by Khandekar et. al. [14]. We make two simple observations referenced in previous work (including [1]):

1. $\text{OPT} \geq \frac{np}{g}$
2. $\text{OPT} \geq \text{OPT}_\infty$

The first is a simple mass bound and the second follows from the fact that OPT is a solution to a more restricted version of the busy time problem than OPT_∞.

3.2.1 4 Approximation for Equal Length Jobs

In this section we give a very simple algorithm which achieves a 4 approximation when all job lengths are equal. Suppose we have n jobs $\mathbb{J} = \{j_1, j_2, ..., j_n\}$ each with their own release times and deadlines but with a common processing time of p.

Our algorithm is simply the following: Solve the $g = \infty$ dynamic program to obtain a temporary schedule. Select any arbitrary job j as the start of a bundle. Successively add jobs it overlaps with to the bundle until the bundle is full (i.e its

height is g). If we cannot create a full bundle, schedule all the jobs on one machine and terminate the algorithm. Otherwise, schedule this full bundle on a machine and repeat.

Because each full bundle is saturated with jobs, it must have at least g jobs in it. So the total number of full bundles cannot exceed $\frac{n}{g}$. Further, because each job has length p, the maximum span of a bundle so formed is at most $3p$. So the total span of these full bundles is at most $\frac{3pn}{g} \leq 3\text{OPT}$. Finally, all the jobs that could not be scheduled in a full bundle can all be scheduled on a single machine. The span of these jobs is at most $\text{OPT}_\infty \leq \text{OPT}$ because of the initial dynamic program. Hence, the 4 approximation result follows.

3.2.2 3 Approximation for Equal Length Jobs

Here, we improve slightly on the previous algorithm to give a 3 approximation when job lengths are equal. The problem setting is the same as in the previous algorithm - we have a set of n jobs $\mathbb{J} = \{j_1, j_2, ..., j_n\}$ with common processing time p.

Consider the set of unscheduled jobs J. Solve the $g = \infty$ dynamic program and obtain the corresponding schedule. Using an adaption of the non-full rigid job technique sketched in the 3 approximation for the active time problem (introduced in [1]), we can represent the span of the unscheduled jobs using a set of jobs such that at most two from the set are live in any given slot. Call this set T. Note that here, there is no preemption and we do not need to consider full and non full slots.

Now every job in $J - T$ overlaps with at least one job in T since otherwise it would have been added to T.

We run our algorithm in two rounds: In the first round, assign each job $j \in J - T$ to a job in T such that the overlap is at least $\frac{p}{2}$. This is depicted in Figure 3.1. Note that such a job must exist since T is contiguous and if a job overlaps less than $\frac{p}{2}$ with one job in T, it will overlap at least $\frac{p}{2}$ with one adjacent to it in the sequence. If a job overlapped with only one job from T, it must have been at the ends and then would have been included in T. Once this assignment is complete, go through each job in T and select at least g from the ones assigned to it to form a bundle. Continue doing so until no longer possible.

Once no longer possible, the first round ends. Now, there are strictly less than g jobs assigned to each job in T but we could still have situations where the depth is more than g (as shown in Figure 3.2). As a part of round two, select g or more such jobs to form another depth g bundle and repeat until no longer possible. Now all the jobs that remain can be scheduled on one machine.

For the analysis, in the first round, because the assigned jobs overlap at least $\frac{p}{2}$ with the job in T, the maximum span of such a full bundle is $\frac{p}{2} + p + \frac{p}{2} = 2p$. In the second round, we form bundles based on regions where the depth is greater than or equal to g. Here, the span can also only be at most $2p$. Similar to the previous analysis, the maximum number of bundles is $\frac{n}{g}$, and therefore, their total span is at most $\frac{2np}{g} \leq 2\text{OPT}$. Finally, the jobs that remained also had a span of at most $\text{OPT}_\infty \leq \text{OPT}$. This gives us the 3 approximation.

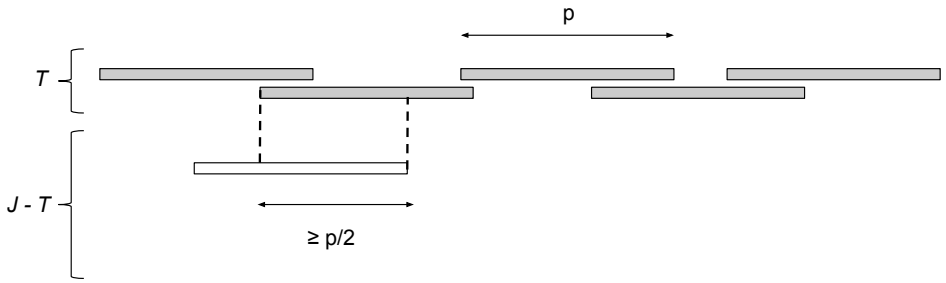

Figure 3.1: 3 approximation for busy time with equal length jobs. The shaded jobs represent the set T whereas the unshaded one is in $J - T$.

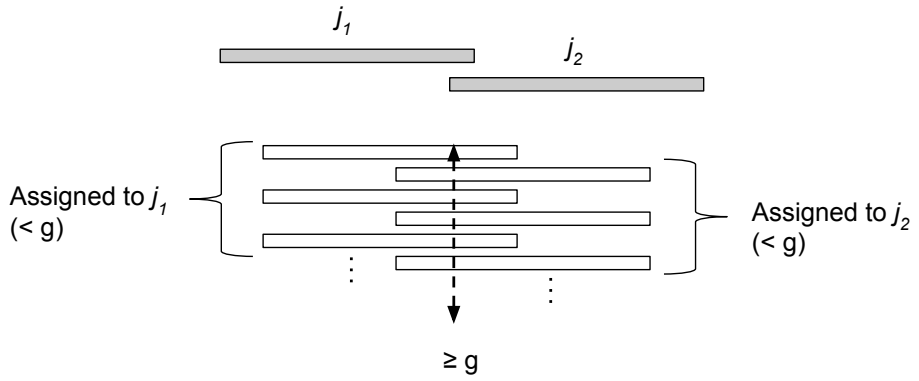

Figure 3.2: An example of how even though the number of jobs assigned to each job in T may be less than g, there may still be regions with overlap greater than g. Here, the overlap from jobs assigned to two adjacent T jobs j_1, j_2 exceeds g.

Chapter 4: Concluding Remarks and Future Work

We summarize the current state of work on the active time problem in Table 4.1. The entries marked by an asterisk were introduced and proved in this thesis. We highlight some of the more interesting research directions for the work described in this thesis:

Complexity Status of the Active Time Problem

This is the most intriguing open question for the active problem. The problem itself sits neatly in the middle of the spectrum with its simpler variants (those restricting the problem to unit length jobs or single thread processing) known to be solvable exactly in polynomial time whereas its generalization to jobs with arbitrary

Algorithm	Upper Bound	Lower Bound
Minimum Feasible Solution	3	3
LP Rounding	2	2
Greedy*	2	2
Local(b)*	?	1.5

Table 4.1: Current State of Work on the Active Time Problem

collections of intervals is known to be NP hard.

Algorithms for the Active Time Problem

This topic is closely related to (and will clearly be influenced by) the previous point. Are approximation algorithms in fact the best we can do? If so, is it possible to achieve something better than our current upper bound of 2? Is a PTAS possible for this problem? Can the local search algorithm be proved to achieve its lower bounds?

Lower Bounds and Algorithms for the Busy Time Problem

The busy time problem is known to be hard. However, given the current state of knowledge about the problem, nothing seems to stop us from coming up with better approximations for the problem. Is 3 the best approximation possible for this problem? Can we derive any lower bounds for the problem?

Bibliography

[1] Jessica Chang, Samir Khuller, and Koyel Mukherjee. Lp rounding and combinatorial algorithms for minimizing active and busy time. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 118–127. ACM, 2014.

[2] Michael R Garey and David S Johnson. *Computers and intractability*, volume 29.

[3] Jessica Chang, Harold N. Gabow, and Samir Khuller. A model for minimizing active processor time. *Algorithmica*, 70(3):368–405, 2014.

[4] Guy Even, Retsef Levi, Dror Rawitz, Baruch Schieber, Shimon Moni Shahar, and Maxim Sviridenko. Algorithms for capacitated rectangle stabbing and lot sizing with joint set-up costs. *ACM Transactions on Algorithms (TALG)*, 4(3):34, 2008.

[5] Philippe Baptiste. Scheduling unit tasks to minimize the number of idle periods: a polynomial time algorithm for offline dynamic power management. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 364–367. Society for Industrial and Applied Mathematics, 2006.

[6] Koyel Mukherjee. *Algorithmic approaches to reducing resource costs in data centers*. PhD thesis, 2013.

[7] Andrew V Goldberg, Sagi Hed, Haim Kaplan, Pushmeet Kohli, Robert E Tarjan, and Renato F Werneck. Faster and more dynamic maximum flow by incremental breadth-first search. In *Algorithms-ESA 2015*, pages 619–630. Springer, 2015.

[8] Andrew V Goldberg, Sagi Hed, Haim Kaplan, Robert E Tarjan, and Renato F Werneck. Maximum flows by incremental breadth-first search. In *European Symposium on Algorithms*, pages 457–468. Springer, 2011.

[9] The Maximum Flow Project. [Online; accessed 03-December-2016].

[10] Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2016. [Online; accessed 07-December-2016].

[11] Jessica Chang. *Energy-aware batch scheduling*. PhD thesis, 2013.

[12] Peter Winkler and Lisa Zhang. Wavelength assignment and generalized interval graph coloring. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 830–831. Society for Industrial and Applied Mathematics, 2003.

[13] Michele Flammini, Gianpiero Monaco, Luca Moscardelli, Hadas Shachnai, Mordechai Shalom, Tami Tamir, and Shmuel Zaks. Minimizing total busy time in parallel scheduling with application to optical networks. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[14] Rohit Khandekar, Baruch Schieber, Hadas Shachnai, and Tami Tamir. Minimizing busy time in multiple machine real-time scheduling. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 8. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.