Algorithms in Throughput Maximization

by

Dylan Vern Phillips Hyatt-Denesik

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

© Dylan Vern Phillips Hyatt-Denesik, 2019

Abstract

Scheduling problems are problems in which jobs are assigned to machines at particular times. A common objective of scheduling is to schedule as many jobs before their deadline as possible, called THROUGHPUT MAXIMIZATION . In this thesis, we provide algorithms for various cases of THROUGHPUT MAXIMIZATION problems, varying bounds on the number of release times, job sizes, deadlines, and even the number of release times the release time and deadlines of a job can span across.

In Chapter 2, we examine the case where both the number of job sizes and the number of release times is bounded by a constant value. The result in this Chapter is progress to closing a known open problem of THROUGHPUT MAXIMIZATION with only constant sized job sizes [20]. The algorithm described in this Chapter is a dynamic program that relies on a framework based around what is known as the Jackson Rule or Earliest Due Date ordering [13].

In Chapter 3, we consider the case of THROUGHPUT MAXIMIZATION where the number of release times and deadlines are bounded by a constant value on a constant number of machines. In this Chapter we find a pseudo-polynomial time exact algorithm that we extend to a PTAS. This algorithm exploits the fact that we can partition the schedule into intervals between two consecutive release times or deadlines, which allows us to assign jobs to each of these intervals in polynomial time. We also show that this case is NP-Complete by a reduction from the PARTITION problem.

In Chapter 4, we present the most general result in this thesis by finding a pseudo-polynomial time algorithm for the case of THROUGHPUT MAXIMIZATION where the only restriction is that there are a constant number of release times, which we extend to the case where the interval between a jobs release time and deadlines has only a constant number of release times, called the span. This algorithm makes use of similar approach to that of Chapter 3, in that we can partition jobs to intervals between consecutive release times and find the schedule interval by interval. The result for the constant span, we use the first

algorithm of this chapter as a subroutine over the maximum span of release times.

No hurdle is too high for a man who simply knows how to adjust

– Gavin Dunne

Acknowledgements

I would like to thank Mohammad Salavatipour for his role as my supervisor and his guidance in my development as a researcher. I am also grateful for his patience in reviewing this text, helping me to revise it and to present the work in a clear and effective manner.

Contents

1 Introduction **1**
1.1 Preliminaries . . . 1
1.1.1 Optimization Problems and Approximation Algorithms . . . 1
1.1.2 Scheduling Problems . . . 3
1.1.3 Knapsack Problems . . . 6
1.2 Problems Considered . . . 7
1.2.1 Terms and Definitions . . . 8
1.3 Prior and Related Work . . . 9

2 Constant Number of Job Sizes and Release Times **11**
2.1 Problem Overview . . . 11
2.1.1 Canonical Schedules . . . 12
2.2 Algorithm . . . 15
2.3 Algorithm Analysis . . . 17

3 Constant Number of Release Times and Deadlines **18**
3.1 Problem Overview . . . 18
3.2 Pseudo-Polynomial Time Algorithm . . . 20
3.2.1 Algorithm . . . 21
3.2.2 Algorithm Analysis . . . 22
3.3 PTAS for Constant Number of Release Times and Deadlines . . . 23
3.3.1 Algorithm . . . 24
3.3.2 Analysis . . . 24
3.4 General Case is Weakly NP-Complete . . . 25
3.5 Extending to a Constant Number of Machines . . . 25

4 Constant Number of Release Times **27**
4.1 Constant number of Release Times . . . 27
4.1.1 Canonical schedule . . . 29
4.1.2 Algorithm . . . 30
4.1.3 Analysis . . . 32
4.2 Constant Span . . . 33
4.2.1 Canonical Schedules . . . 34
4.2.2 Algorithm . . . 35
4.2.3 Algorithm Analysis . . . 35
4.2.4 Extending to a Constant Number of Machines . . . 36

5 Conclusion **38**

References **40**

Appendix A Multiple Knapsack Problem for Uniform Profits **42**

List of Figures

2.1 In this example we schedule jobs $\{1, 2, 3, 4, 5, 6\}$ in window $[s_k, s_{k+1}]$, with start times in release interval $\sigma(k) = i$. The pale colored jobs represent Y_i since they all have deadline less than s_{k+1}. The grey jobs represent X_i since they have deadline at least s_{k+1}. Note that job 6 is scheduled such that it spans 3 release times, r_{i+1}, r_{i+2}, and r_{i+3}. 14

3.1 In this example the grey blocks represent allotments and the white blocks represent the straddle jobs of a canonical schedule. 21

4.1 In this example, the dark blocks are the straddle jobs (j_i, s_i) and (j_{i+1}, s_{i+1}), and the light grey blocks are the allotments. The jobs $\{1, 2, 3, 4\}$ are the jobs with deadline in this interval and are scheduled before the allotments in increasing order of deadline and right-shifted to the starting time of the allotments, which is denoted L_i. 30

Chapter 1

Introduction

1.1 Preliminaries

We first provide a summary of terms and concepts used in this thesis. The definitions here are from [4], [22], and [23].

1.1.1 Optimization Problems and Approximation Algorithms

Decision Problems and NP-Completeness

A *decision problem* is a problem that has either a "yes" or a "no" as an answer. Decision problems can be viewed as languages over the alphabet $\{0,1\}^*$. The language corresponding to a decision problem is the set of binary strings that encode yes instances of the problem.

NP is the class of languages such that for each language $L \in \mathbf{NP}$ there are a pair of polynomial functions p and q, and a deterministic Turing Machine M such that for every string $x \in \{0,1\}^*$ one of the following is true.

- if $x \in L$, then there exists a string $y \in \{0,1\}^*$, called a *certificate*, with length at most $p(|x|)$ such that $M(x,y)$ accepts in at most $q(|x|)$ many steps.
- if $x \notin L$, then for any string y with length at most $p(|x|)$, $M(x,y)$ rejects in at most $q(|x|)$ many steps.

Let L_1 and L_2 are two languages in **NP**, L_1 is said to *reduce* to L_2 if there is a Turing machine $M : \{0,1\}^* \rightarrow \{0,1\}^*$ that maps a string x to a string $y \in L_2$ if and only if $x \in L_1$, that takes a polynomial in x number of steps. A language L_1 is called **NP**-hard if for every language $L_2\mathbf{NP}$, L_1 reduces to L_2. A language is called **NP**-Complete if it is both in the class **NP** and **NP**-hard.

An **NP**-Complete language L is called *strongly* **NP**-Complete if every language in **NP** can be polynomially reduced to L in such a way that the values of in the reduced instance are always written in unary. If L is not strongly **NP**-Complete then it is *weakly* **NP**-Complete.

Optimization

An **NP**-*optimization problem*, Π, consists of

- A set of valid instances, D_Π, where we can recognize if an instance $I \in D_\Pi$ in time polynomial in $|I|$. We assume that numbers of the instance I are written as binary strings, and therefore any number that is exactly specified is rational.
- Each instance $I \in D_\Pi$ has a set of *feasible solutions*, $S_\Pi(I) \neq \emptyset$. We assume that every solution $s \in S_\Pi(I)$ is polynomially bounded in length in $|I|$, and that there is a polynomial time algorithm that decides if $s \in S_\Pi(I)$ given the pair (I, s).
- There is an objective function obj_Π, that assigns a nonnegative rational value to each (I, s) pair, where I is an instance of Π and s is a feasible solution of I. This function is computable in polynomial time, and can also be referred to with more common terms such as *cost, length, weight* etc.

An *optimal solution* of a maximization (minimization) optimization problem, Π, is a feasible solution that achieves the largest (smallest) objective function value. We let $\text{OPT}_\Pi(I)$ denote the objective function value of an optimal solution to instance I, which we will denote as OPT when it is clear from context that we mean the optimal of an instance.

For an **NP**-optimization problem there is an obvious decision problem that can be associated by simply finding a bound on the optimal solution. The decision version of an **NP**-optimization problem Π is an instance I of Π and a rational number B, which we denote by the pair (I, B). If we have a maximization (minimization) problem Π, then we can define the decision version of the problem to have a "yes" answer if and only if there is a feasible solution to I with cost at least B (at most B). So we say that (I, B) is a "yes" instance to the problem if the bound is satisfied, and a "no" instance otherwise.

If we can compute the optimal solution for Π in polynomial time, then we can compare this solution to B and solve the decision version of the problem. What this means is that if we can establish hardness for the decision problem, then the hardness is inherited by Π. If the decision version of a problem is **NP**-hard then we can abuse notation to say that the optimization version is **NP**-hard as a shorthand when saying that it is the optimization version of an $\mathbf{NP} - hard$ problem.

Approximation Algorithms

Let Π be a maximization (minimization) problem, and let δ be a function, $\delta : \mathbb{Z}^+ \to \mathbb{Q}^+$, with $\delta \leq 1$ ($\delta \geq 1$). An algorithm $\mathcal{A}$ is said to be a *factor δ approximation algorithm for* Π if, its running time is bounded by a polynomial in $|I|$, and, for every instance of I, $\mathcal{A}$ gives a feasible solution $s \in S_\Pi(I)$ such that $\text{obj}_\Pi(I, s) \geq \delta(|I|) \cdot \text{OPT}(I)$ ($\text{obj}_\Pi(I, s) \leq \delta(|I|) \cdot \text{OPT}(I)$). We say that δ is the *approximation ratio* of $\mathcal{A}$.

We say that an algorithm $\mathcal{A}$ is an *approximation scheme* for **NP**-hard optimization problem Π if, given an instance I of Π and an approximation parameter $\varepsilon > 0$, it outputs a solution s such that $\text{obj}_\Pi(I, s) \leq (1 + \varepsilon)\dot{O}PT$ if Π is a minimization problem and, $\text{obj}_\Pi(I, s) \geq (1 - \varepsilon)\dot{O}PT$ if Π is a maximization problem. $\mathcal{A}$ is said to be a *polynomial time approximation scheme*, or PTAS for short, if its runtime is bounded by a polynomial in $|I|$ for every fixed $\varepsilon > 0$.

The definition for a PTAS only requires that the runtime is bounded by a polynomial in $|I|$, but may depend arbitrarily on ε. If we have an algorithm $\mathcal{A}$ that satisfies the conditions of a PTAS, but has runtime bounded by a polynomial in both $|I|$ and $1/\varepsilon$, then we say that $\mathcal{A}$ is a *fully polynomial time approximation scheme.*

Reductions

Let Π_1 and Π_2 be two decision problems. We say that there is a *polynomial time reduction* from Π_1 to Π_2, often said that Π_1 reduces to Π_2 in polynomial time, if there is a polynomial-time algorithm $\mathcal{A}$ that takes instance I_1 of Π_1 to $I_2 = \mathcal{A}(I_1)$ which is an instance of Π_2, such that I_1 is a yes instance of Π_1 if and only if I_2 is a yes instance of Π_2.

We make use of the hardness of the PARTITION problem later on. The PARTITION problem can be written as

Definition 1.1.1 (Partition)**.** *Given n positive integers $s_1, \ldots, s_n$, is there a subset $J \subseteq I = \{1, \ldots, n\}$ such that*

$$\sum_{i \in J} s_i = \sum_{i \notin J} s_i?$$

We will use the well known hardness result due to Karp [15] we rely on.

Theorem 1.1.1. *The* PARTITION *problem is weakly* ***NP****-complete*

1.1.2 Scheduling Problems

A common type of optimization problem and one that will be central to this thesis is that of creating a schedule. There are n jobs, each with index $j \in \{1, \ldots, n\}$, that must be scheduled on m *machines* $M_i (i = 1, \ldots, m)$. Each job j has a *release time* r_j, which is the first time point that it is available for processing. Each job j may be associated with a specific subset of machines $\tilde{M} \subseteq \{M_1, \ldots, M_m\}$, where j may only be scheduled on machines from $\tilde{M}$. If $\tilde{M}$ is equal to the set of all machines for each job then we say that that the machines are *parallel*, and *dedicated* machines otherwise.

Jobs possess a *cost function* $f_i(t)$, which measures the cost of completing job J_i at time time t. Cost functions can be computed solely based on time, but can also account deadlines d_j and weights into their computation w_j. In general, we assume that the values p_j, r_j, d_j, and w_j are all integer, across any range of numbers $a_1, \ldots, a_n$ we set $a_{\max} = \max_i a_i$ and

$a_{\min} = \min_i a_i$, so we understand the meaning of terms like $r_{\max}$ and $p_{\max}$. A schedule is called *feasible* if the time allocated to separate jobs does not overlap, if two time intervals allocated to a single job do not overlap, and if it meets any problem-specific characteristics.

Scheduling problems can be classified by the differences in machine environments, job characteristics, and optimality criterion. These variants of scheduling problems can be specified in a 3-field notation $\alpha|\beta|\gamma$ introduced by Graham et al. [10], where α is a string describing the machine environment, β is a string describing the job characteristics, and γ denote the optimality criterion.

Job Characteristics

The job characteristics string is often specified by up to 6 string elements $\beta_1, \beta_2, \beta_3, \beta_4, \beta_5$, and β_6 as in [4].

The string element β_1 indicates whether *preemption* is allowed, which, if allowed, means that a job that is processing may be interrupted and resumed at a later time, even on another machine. If preemption is allowed then $\beta_1 = pmtn$, otherwise β_1 does not appear in β. Note, that a job may be interrupted and resumed several times.

The string element β_2 describes precedence relations between jobs. Precedence relations can be represented by an acyclic digraph $G = (V, A)$ where $V = \{1, \ldots, n\}$ corresponds to the jobs and there is edge $(i, k) \in A$ if and only if job j_i must be completed before job j_k. In this case we set $\beta_2 = prec$, otherwise it does not appear in β.

The string element β_3 describes the nature of release times for jobs. We let $\beta_3 = r_j$ if each job has its own release time, and $\beta_3 = r$ if all jobs share a release time r, though in this case β_3 is often omitted as it does not change the problem. If β_3 is equal to $\#r_j \in O(1)$ then the number of unique release times is bounded by some constant value. The interval between consecutive unique release times is called a *release interval*.

Restrictions on processing times are denoted by β_4. If all jobs have identical processing time then β_4 is written as $p_j = p$, and is $p_j = 1$ to denote the specific case of unit processing times. The number of unique processing times is denoted by $\#p_j$, so if the number of unique processing times is bounded by a constant then β_4 is equal to $\#p_j \in O(1)$. The value of β_4 could be written in another way if the characteristic implied is obvious such as $p_j \in \{1, 2\}$, which states that processing times are either 1 or 2.

Job deadlines are described by string element β_5, which is equal to d_j if each job j has a deadline, and it's equal to $\#d_j \in O(1)$ if the number of unique deadlines is bounded by a constant. One can also use β_5 to describe the relation between release times and deadlines, we use $span_j$ to denote the number of release times between job j's release time and deadline.

If jobs must be grouped into batches and scheduled together then the string β_6 is used.

A batch in this case is a group of jobs that are scheduled on a single machine concurrently, with the completion time of the batch being equal to the completion time of the longest job.

Machine Environment

The machine environment is characterized by a string $\alpha = \alpha_1\alpha_2$, where α_1 describes the type of machines, and α_2 is an integer describing the number of machines. The string α_1 can be from the set $\{\circ, P, Q, R\}$ where $\circ$ is the empty symbol. The case that $\alpha_1 = \circ$ means each job must be scheduled on a specified machine. The value for α_2 can be an integer value or $\circ$, with $\circ$ indicating that the number of machines is not specified by the scheduling problem. A machine environment that we care about in this work is $\alpha_1 = circ$ and $\alpha_2 = 1$, which denotes that all jobs may be scheduled on the same machines

The machine α_1 strings environments $\{P, Q, R\}$ are some of the most common machines environments, denoting different types of parallel machines. That is, each job can be processed on any one of the available machines $M_1, \ldots, M_m$. If $\alpha_1 = P$ then we say that the machines are **identical and parallels**, so the processing time of job j is p_j on each machine. If $\alpha_1 = Q$, then the machines are **uniform and parallel**. That is, each job j has processing time p_j/s_i on machine M_i, where s_i is the *speed* of a machine. If $\alpha_1 = R$, the speed of machine is **independent and parallel** to all other machines. That is, job j has processing time $p_j/s_{i,j}$ on machine M_i, where $s_{i,j}$ is the speed of job j on machine M_i.

Optimality Criterion

The **finishing time** of job j is denoted by C_j, and the associated **cost** function $f_j(C_j)$. There are two common types of overall schedule cost functions

$$f_{\max}(C) := \max\{f_j(C_j) | j = 1, \ldots, n\}$$

and

$$\sum f_j(C) := \sum_{j=1}^{n} f_j(C_j)$$

which are referred to sometimes as bottleneck objectives and sum objectives respectively. The problem is thus to find a schedule that maximizes or minimizes the total cost function.

The most common objective function considered is the **makespan** $\max\{C_j | j = 1, \ldots, n\}$ problem, with the goal of minimizing this value.

Other objective functions can depend on the deadlines associated to each job. Some objectives include for each job j are: **lateness**, $L_j := C_j - d - j$; **tardiness**, $T_j := \max\{0, C_j - d_j\}$; and **throughput**, $U_j := 1$ if $C_j \leq d_j$ or 0 otherwise. If the objective function is $\sum_j U_j$ or $\sum_j w_j U_j$ then the objective is often called *throughput maximization*, and it is common to ignore the string d_j in the job characteristic portion of the Graham representation of the problem since the problem would be trivial without job deadlines.

Schedule Ordering In a Single Machine Instance

The single machine case of scheduling has been the subject of research since early work due to Jackson [13]. It is interesting to note that for single machines problems, in the case that $r_j = 0$ for each job and the objective function is monotone with respect to job finishing times, then only schedules without preemption and without idle time need to be considered. This fact follows since the optimal objective function does not increase if preemption is introduced.

Consider a schedule where job j is scheduled preemptively in the two intervals $[t_1, t_2)$ and $[t_3, t_4)$, where $t_1 < t_2 < t_3 < t_4$. We can reschedule the part of j that is in $[t_1, t_2)$ to be between $[t_3 - (t_2 - t_1), t_3)$, with anything originally scheduled in $[t_3 - (t_2 - t_1), t_3)$ getting pushed back $t_2 - t_1$ units of time. Due to the monotonicity of the objective function, if we repeat this argument for every instance of preemption we can eliminate preemption with no loss to the objective function.

Another useful property of the single machine environment with shared release time job characteristic is the following due to Jackson [13]:

Theorem 1.1.2 (Jackson Rule)**.** *In the case of all jobs sharing a release time, re-scheduling any feasible schedule in non-decreasing order of deadline results in a feasible schedule with the same throughput.*

Another common name for this result is the Earliest Due Dates rule or EDD rule. We say that jobs are scheduled according to the Jackson rule if they are scheduled in non-decreasing order of deadline.

An early application of the Jackson Rule was in *Moore's Algorithm* [19], which says that given n jobs and a single machine, $r_j = 0$ for all j there is a polynomial time algorithm for maximum throughput. The algorithm is to sort the jobs in non-decreasing order of processing time, and schedule them in that order until a job is late, or every job has been considered. If job j is late, then reschedule the jobs scheduled and j according to the Jackson Rule, if a job is still late when rescheduled then we do not schedule j, and let the other jobs be scheduled in non-decreasing order of processing time. If every job has been considered, then reschedule the jobs according to the Jackson Rule.

1.1.3 Knapsack Problems

In an instance of the typical KNAPSACK PROBLEM, we are given a set of items $N = \{1, \ldots, n\}$, each item $j \in N$ has a profit p_j and a weight w_j, and a knapsack which has a capacity c. The most typical objective of this problem is to pick out a subset $\tilde{N} \subseteq N$ such that the sum of item weights in $\tilde{N}$ does not exceed the capacity c, but the sum of item profits in $\tilde{N}$ is maximized.

We are interested in the generalization to the Knapsack Problem known as the Multiple Knapsack Problem (MKP) where we assume that there are multiple knapsacks $M_i, i = 1, \dots, m$, where each knapsack M_i has positive capacity c_i. The goal in this case is to find m disjoint subsets $\tilde{N}_i \subseteq N$, $i = 1, \dots, m$, where the items of each $\tilde{N}_i$ are assigned to knapsack M_i, such that the sum of item weights does not exceed c_i, with the objective of maximizing the sum of item weights from all $\tilde{N}_i$ subsets.

The MKP is known to be strongly NP-hard and thus does not admit a pseudo-polynomial time algorithm. However, if we assume that items have uniform profit, then the problem has a very simple pseudo-polynomial algorithm that we show in Appendix A. If the items are not assumed to possess a uniform profit then Chekuri et al. [6] showed that this problem admits a PTAS.

1.2 Problems Considered

All problems in this thesis are considered with the throughput maximization optimality criterion, the jobs must be scheduled non-preemptively, each has a release time and deadline, with uniform weight.

Constant Number of Job Sizes and Release Times

In this problem, we assume that there are a constant number of processing times or job sizes, and a constant number of release times. This problem is written using Graham notation as $1|\#p_j \in O(1), \#r_j \in O(1)| \sum_j U_j$. We explore this problem in Chapter 2, where we make use of the constant number of processing times to classify jobs to find an exact solution in polynomial time. We will let P denote the number of unique processing times and R denotes the number of unique release times.

Our first result is a step towards closing a known open problem [20] that says it is unknown if there is a polynomial time solution to $1|r_j, p_j < C| \sum_j U_j$ by relaxing the restriction on processing times but increasing restrictions on release times.

Theorem 1.2.1. *There is a dynamic program that finds an exact solution to* $1|\#r_j \in O(1), \#p_j \in O(1)| \sum_j U_j$ *that is polynomial in* n.

Constant Number of Release Times and Deadlines

In Chapter 3 we examine the case of Throughput Maximization where we assume there are a constant number of unique release times and deadlines, which we write using the 3-field notation as $Pm\big|\#r_j \in O(1), \#d_j \in O(1)\big| \sum_j U_j$. We let R represent the number of unique release times and D represents the number of unique deadlines. In this Chapter we will demonstrate the algorithm on a single machine, and show that the algorithm can easily be extended to a constant number of machines. The methods we develop in Chapter 3 to

solve this problem will serve as groundwork for the methods in Chapter 4 where make fewer assumptions about the number of release times and deadlines.

Theorem 1.2.2. *There is an algorithm that finds an exact solution to $Pm|\#r_j \in O(1), \#d_j \in O(1)|\sum_j U_j$ that is polynomial in both the number of jobs and $p_{\max}$*

Furthermore, this case admits a PTAS.

Constant Number of Release Times

In Chapter 4 we examine the case where there are only a constant number of unique release times and a constant number of parallel machines, which can be written using the 3-field notation as $Pm|\#r_j \in O(1)|\sum_j U_j$. The algorithm presented in this Chapter generalises the one found in Chapter 3, taking extra care to handle the fact that the number of job deadlines is not constant.

Theorem 1.2.3. *There is an algorithm that gives an exact solution to $Pm|\#r_j \in O(1)|\sum_j U_j$ and runs in time polynomial in n and $p_{\max}$.*

This is done with techniques similar to basic knapsack packing algorithms and is where the running times dependence on $p_{\max}$ comes from. This leaves the problem of finding a PTAS open.

Constant Span

The last and most general case of THROUGHPUT MAXIMIZATION we examine in this Thesis is the case where the maximum span of job release times and deadlines is bounded by a constant, which we write in the 3-field notation as $Pm|span_j \in O(1)|\sum_j U_j$. The algorithm for this problem uses the algorithm for $Pm|\#r_j \in O(1)|\sum_j U_j$ as a subroutine over the maximum span of any job. This gives the following Theorem.

Theorem 1.2.4. *There is an algorithm that finds an solution to $Pm|span_j \in O(1)|\sum_j U_j$ and runs in time polynomial in n and $p_{\max}$.*

1.2.1 Terms and Definitions

A schedule in the preceding problems is said to be feasible if, in addition to the usual requirements for feasibility, all jobs scheduled with completion time after their deadlines are scheduled after all jobs scheduled with completion time at most their deadlines. This is because we are optimizing the throughput of the schedule, so any job that is not scheduled on time can be ignored for the purposes of future scheduling as it no longer affects the throughput. We thus say that a job whose completion time exceeds its deadline is "not scheduled". Furthermore, we can say that jobs are *available* at a time point t if they are not scheduled by time t, t is less than their deadline.

For a fixed schedule S, we say that a point p is *straddled* in this schedule if there is a job whose scheduled start time and finish time are less than and greater than p, respectively.

A schedule is called *left-shifted* if every job starts either at their release time or at the completion time of another job. We say that we are "left-shifting" a schedule if the order of jobs in the schedule is preserved but the jobs are rescheduled to possibly earlier times so that the schedule becomes left-shifted.

In general, we will enumerate unique release times and deadlines in increasing order, $r_1 < r_2 < \cdots < r_{\max}$ and $d_1 < d_2 < \cdots < d_{\max}$ respectively. Note that this allows us to call the interval between r_i and r_{i+1} release interval i. When it is clear from context if we have a job j then we may say that it releases at time r_j and has deadline d_j and size p_j.

1.3 Prior and Related Work

One of the oldest studied versions of throughput maximization is a single machine with uniform job profits, $1||\sum_j U_j$. Jackson [13] showed that if all jobs can be scheduled before their deadlines then a schedule for $1||\sum_j U_j$ can be found by applying what is now known as the Jackson Rule. This problem was also considered by Held and Karp [11] who found a general dynamic program for scheduling problems with cost function equal to the sum of costs dependent on each job completion time. However, this algorithm was exponential in the number of jobs, since the table was indexed by all possible subsets of jobs. An exact $O(n^2)$ time solution to this was later found by Moore [19], who made repeated use of the Jackson Rule to get an exact solution. This algorithm by Moore will be used in Chapter 2.

The problem $1||\sum_j w_j U_j$ is known to be weakly NP-complete [15][18], and recieved a pseudo-polynomial time algorithm from Lawler et al. [17], which was extended to a PTAS by Karger [14]. If the weights and release times are oppositely ordered in the sense that if $r_1 \leq \cdots \leq r_n$ then $w_1 \geq \cdots \geq w_n$, then Lawler [16] finds an exact solution in polynomial time.

With the addition of release times the problem, $1|r_j|\sum_j U_j$, becomes Strongly NP-hard [9]. However, Kise et al. provide a polynomial time solution to a special case of this problem if the release times and deadlines are agreeable in the sense that $r_1 \leq \cdots \leq r_n$ implies $d_1 \leq \cdots \leq d_n$. A simple greedy algorithm by Spieksma [21] has been shown to give a 1/2-approximation to this problem. Bar-Noy et al. [3] give a 1/2-approximation to $1|r_j|\sum_j w_j U_j$ by using time indexed linear programming relaxations of fractional schedules, as well as a $1 - \frac{1}{(1+1/m)^m}$-approximation to $P|r_j|\sum_j w_j U_j$ where m is the number of machines. Chuzoy et al. [7] improve this result for the restricted case $P|r_j|\sum_j w_j U_j$ to $1 - 1/e$. The current best result is a 0.644-approximation algorithm to $Pm|r_j|\sum_j U_j$ due to Sungjin et al. [12], who also give a $1 - O(\sqrt{\log(m)/m})$-approximation to $P|r_j|\sum_j U_j$ which also holds for the weighted case if the largest deadline deadline is bounded by a polynomial.

A special case of throughput maximization that has received a great deal of study is for equal length jobs. This was first considered by Baptiste [1], who closed an open problem by finding a polynomial time solution to $1|r_j, p_j = p|\sum_j w_j U_j$ by a dynamic program. Baptiste [1] also introduced the notion of *left-shifting* which is used extensively throughout this thesis. Baptiste then extends the methods use in this result in [2] to find an exact dynamic programming solution to $Pm|r_j, p_j = p|\sum_j w_j U_j$ by applying a clever resource profile vector that bounds the range per machine that jobs considered can be both released in and scheduled in. The problem where the number of parallel machines is given as part of the input is first examined by Brucker et al. in [5] where a simple polynomial time algorithm is given for $P|p_j = p|\sum_j w_j U_j$ by greedily scheduling jobs based on deadline, swapping late jobs with earlier on time jobs of less profit. The more general problem including deadlines is shown to be solvable in polynomial time by Brucker in [4] with a network flow algorithm, a faster algorithm was given by Dourado et al. [8].

Chapter 2

Constant Number of Job Sizes and Release Times

In this Chapter we examine the problem of THROUGHPUT MAXIMIZATION with a constant number of unique release times and job sizes on a single machine, which is written in the Graham notation as $1|\#r_j \in O(1), \#p_j \in O(1)| \sum_j U_j$. The number of unique job sizes is P and the number of unique release times is R. We will provide a dynamic programming algorithm in this Chapter to get the following theorem.

Theorem 2.0.1. *There is a dynamic programing algorithm that finds an exact solution to $1|\#r_j \in O(1), \#p_j \in O(1)| \sum_j U_j$ that is polynomial in n.*

2.1 Problem Overview

Recall that for a fixed schedule S, a point p is *straddled* if there is a job whose scheduled start time and completion time are less than and greater than p respectively. A key point of the algorithm will be to schedule sets of jobs with start time between two release times so that at most one job straddles a release time.

To provide intuition for the algorithm presented in this chapter, we simplify the explanation by assuming that release times are not straddle by any jobs. With this assumption, the jobs with start time in release interval i, are scheduled entirely in release interval i. That is, every job will start at no sooner than r_i, and will end no later than r_{i+1}. This assumption is useful because even in the case that there are straddled release times, the intuition still holds, there is just some extra checking required to schedule jobs between them, the difficulty lies in scheduling jobs between them.

We can thus apply Theorem 1.1.2 to any such schedule to feasibly re-schedule the jobs in release interval i in non-decreasing order of deadline. This new ordering of jobs partitions them into the jobs with deadline less than r_{i+1}, which we call Y_i, and the jobs with deadline greater than or equal to r_{i+1}, which we call X_i. Note that every job in Y_i is scheduled before

every job in X_i. For any feasible schedule we can repeat this argument for every release interval. Our algorithm is to find the X_i and Y_i for every release interval i for an optimal schedule.

We build the schedule one release interval at a time, starting from release interval number one. This is so that when a job is scheduled in an earlier release interval it isn't considered for later ones. To find a schedule for release interval i, we assume that there is a schedule in release intervals 1 to $i-1$. We start by guessing the jobs in an optimal X_i and schedule them in order of deadline right-shifted to r_{i+1}.

To find Y_i we consider every job with deadline in the interval (r_i, r_{i+1}) that is not already scheduled. We consider these jobs to all have been released at time r_i and have deadline no later than the start time of jobs in X_i and apply Moore's algorithm [19]. Assuming we correctly guessed the optimal X_i, then this gives an optimal schedule in release interval i.

To make a guess for the optimal X_i we will guess an integer vector $\{x_t^i\}_{t \in P}$, where x_t^i is the number of jobs in X_i of size t. We will pick the first x_t^i jobs of size t with release time at most r_i, and deadline at least r_{i+1} in order of deadline. That is, we will pick the jobs available for X_i with the earliest deadline, and we say that a job picked in this way is "in" X_i.

The remaining difficulty is to deal with jobs that straddle release times. We will deal with this by guessing some amount of *slack*, s for the last job in X_i to straddle r_{i+1}. Once we guess this slack time, we can find jobs for X_i from those with deadline at least this slack time instead of from those with deadline at least r_{i+1}. Since the release time r_i might also be straddled to some time s' we can view these jobs as being scheduled in the window $[s, s']$. We will view the schedule as a sequence of these windows. We see will an example of these windows in Figure 2.1.

2.1.1 Canonical Schedules

Our approach is to find a schedule for release intervals in increasing order, starting from the earliest release time r_1, to the last release time $r_{\max}$. To see how to find these schedules, we will take some optimal schedule and reschedule the jobs of it to adhere to a certain structure. This structure will define so-called *canonical* schedules, allowing us to enumerate across all possible canonical schedules in order to find an optimal solution.

The schedules that we will consider from here on are left-shifted unless otherwise specified, in particular we assume canonical schedules are left-shifted. By the definition of a left-shifted schedule, the start time of every job is either the end time of another job or a release time. Therefore, we can partition the jobs into continuous intervals of jobs whose leftmost points are release times. We call the set of possible rightmost points of these intervals *slack times*.

Definition 2.1.1 (Slack times)**.** *Let slack times $\mathcal{T}$ be the set of points t such that there is a release time $r_i \in R$, and a subset of jobs $\mathcal{I} \subseteq \mathcal{J}$, such that $t = r_i + \sum_{j \in \mathcal{I}} p_j$*

We begin by taking an optimal schedule $\mathcal{O}$. As we have seen, since $\mathcal{O}$ is left-shifted it is a series of intervals with jobs continually scheduled starting from release times and ending at slack times. This observation will be useful for describing *windows* that we will schedule jobs in.

Consider release interval i in $\mathcal{O}$, the left-most starting time of a job in this interval is a slack point we call s. The right-most end time of a job that starts in this interval is also a slack time, which we call s'. Because $\mathcal{O}$ is assumed left-shifted, if $s \neq r_i$ then s is the endpoint of a job straddling r_i, otherwise it is clear that $s = r_i$. Similarly, either $s' = r_{i+1}$ or s' is the end time of a job straddling r_{i+1} since we require that it is the end time of a job starting in the interval.

Definition 2.1.2. *Given slack times s and s', the window for release interval i is $[s, t]$ if s is at least r_i, and t is the maximum of r_{i+1} and s'.*

An important note is that while we say that the window is defined for a particular release interval, it may span many release intervals, this reflects the fact that a job may start in a release interval and straddle many release times. We define windows in this way so that a schedule can be viewed as a series of windows scheduled with no idle time between them. To enumerate the windows of a schedule, we number them left to right in order as $\{[s_k, s_{k+1}]\}_{k=1}^{k_{\max}-1}$, where s_1 is always equal to r_1, and $k_{\max}$ is the index of the last window. For a sequence of windows we can find the release interval a window $[s_k, s_{k+1}]$ is defined for with the function $\sigma : \mathbb{Z} \to \mathbb{Z}$, which maps k to the index of the most recent release time less than s_k. So window $[s_k, s_{k+1}]$ is the window for release interval $\sigma(k)$.

For $\mathcal{O}$, in increasing order of release interval we sort and reschedule the jobs starting in the interval according to deadline by the Jackson Rule 1.1.2. Note that this may cause some jobs to start after the release interval, which is why we perform this sorting in increasing order of release intervals. For window $[s_k, s_{k+1}]$, we let $X_{\sigma(k)}$ denote the scheduled jobs with deadline greater than or equal to s_{k+1}, and $Y_{\sigma(k)}$ denote the scheduled jobs with deadline less than s_{k+1}. An example of the jobs scheduled in a window is given in the Figure 2.1.

We want to show that in window $[s_k, s_{k+1}]$ for $\mathcal{O}$, given a schedule for windows 1 to $k-1$, the jobs in $X_{\sigma(k)}$ can be represented by an integer vector $\{x_t^{\sigma(k)}\}_{t \in P}$, where $x_t^{\sigma(k)}$ is the number of jobs of size t in $X_{\sigma(k)}$. To show this we will first need the following lemma, which shows we can swap jobs of equal size but different deadlines with no loss to throughput.

Lemma 2.1.1. *Suppose we have a feasible schedule and job j is scheduled at time t, j' is another job of same size as j that is also available at time t, and that $t + p_j \leq d_{j'} \leq d_j$.*

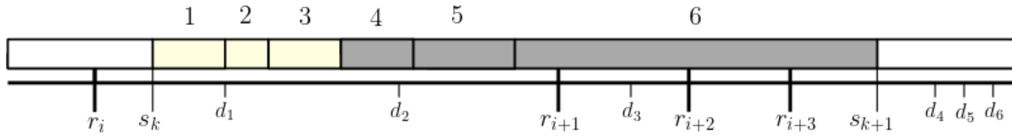

Figure 2.1: In this example we schedule jobs $\{1, 2, 3, 4, 5, 6\}$ in window $[s_k, s_{k+1}]$, with start times in release interval $\sigma(k) = i$. The pale colored jobs represent Y_i since they all have deadline less than s_{k+1}. The grey jobs represent X_i since they have deadline at least s_{k+1}. Note that job 6 is scheduled such that it spans 3 release times, r_{i+1}, r_{i+2}, and r_{i+3}.

Then if j' is scheduled after j, or not scheduled at all, then we can swap where the two jobs are scheduled with no loss of throughput.

Proof. We know that $d_{j'} \geq t + p_j$, so we can schedule j' at time t on time since $t + p_{j'} = t + p_j \leq d_{j'}$ since we assume $p_j = p_{j'}$. We consider the cases of whether j' is initially scheduled or not.

Suppose j' is not scheduled, then to swap j' and j we schedule simply j' at time t and don't schedule j, giving a schedule with equal throughput.

Suppose instead that j' is scheduled after j at time $r > t$. Since $p_j = p_{j'}$, we know $r + p_j \leq d_{j'} < d_j$, which means we can feasibly schedule j at time r with no loss of throughput. □

Assuming that there is already assigned schedules for windows 1 to $k - 1$, then from among the jobs available at time s_k that have deadline at least s_{k+1} we take the $x_t^{\sigma(k)}$ jobs of type t with earliest deadline, we say that jobs picked this way are picked *greedily*. We can use Lemma 2.1.1 to swap jobs found greedily with the jobs of type t originally in $X_{\sigma(k)}$ in $\mathcal{O}$. This swapping is done in increasing order of deadline for both the jobs greedily picked and the jobs originally in $X_{\sigma(k)}$, so the first job scheduled in $X_{\sigma(k)}$ is swapped with the greedily picked job with earliest deadline, and so on. We can use this procedure to show the following Lemma, which will give us a key structure to use in our algorithm.

Lemma 2.1.2. *There is an optimal schedule where in each window $[s_k, s_{k+1}]$, the jobs in $X_{i(k)}$ are picked greedily. Moreover, the jobs scheduled in the window are scheduled in non-decreasing order of deadline.*

Proof. We will take an optimal schedule $\mathcal{O}$ and prove the claim by induction on the index of the window $[s_k, s_{k+1}]$. To simplify notation, we let $\sigma(k) = i$, the release interval the jobs in window k will be scheduled to start in. We first denote the number of each job type in X_i by the integer vector $\{x_t^i\}_{t \in P}$, the number of jobs of each type t. If we assume that the claim holds for each window before window k, then we can *swap* the jobs initially in X_i with those found *greedily* according to deadline as above. Every time we swap jobs we use

a *sorting* procedure that sorts the jobs in the window in order of deadline, then shrink the window to not include jobs that start outside of the window.

We first *swap* the jobs originally in X_i with the jobs greedily picked from those with deadline at least s_{k+1}, and call these jobs X_i.

The *sorting* procedure is always applied after a *swap* procedure. We reschedule the jobs of X_i and Y_i in non-decreasing order of deadline by the Jackson Rule 1.1.2. If, as a result of this sorting, some jobs are scheduled with start time at least r_{i+1}, we remove these jobs from X_i and Y_i, and we reduce the size of the window to reflect this loss of jobs. So we change s_{k+1} to be the end time of the last job now with start time less than r_{i+1}. Since we have shortened the size of the window, the jobs in Y_i may now have deadline at greater than or equal to s_{k+1}, so we move these jobs from Y_i to X_i.

We can then repeatedly perform these procedures until the window size does not change, since we will have the jobs scheduled in order of deadline and X_i picked greedily. We know this will terminate since the window can shrink only a finite number of times.

□

With this lemma we know that there is an optimal schedule $\mathcal{O}$ that is a left-shifted sequence of windows $\{[s_k, s_{k+1}]\}_{k=1}^{k_{\max}}$, where in each window the jobs are scheduled in non-decreasing order of deadline and all jobs start in the release interval $[r_{i(k)}, r_{i(k)+1}]$. These jobs are partitioned into the jobs with deadline less than s_{k+1}, namely $Y_{i(k)}$, and the jobs with deadline at least s_{k+1}, namely $X_{i(k)}$. Finally, we know $X_{i(k)}$ can be represented by an integer vector and the jobs can be selected greedily as in Lemma 2.1.1. We say that *any* schedule that satisfies these conditions is a *canonical* schedule.

2.2 Algorithm

Our algorithm is a dynamic program that finds an optimal canonical schedule by searching across schedules that have the same properties as a canonical schedule. This will be done by constructing the schedule one window at time in a left to right manner, by guessing both the size of optimal windows and guessing the optimal choice of X_i's for each of these windows.

Given the previous windows for the schedule, $\{[s_k, s_{k+1}]\}_{k=1}^{K-1}$, and their corresponding vectors $\{X_{\sigma(k)}\}_{k=1}^{K-1}$, we need to know exactly what jobs are available at time s_K. To find these jobs, starting from $[s_1, s_2]$ in increasing order of windows, we find the jobs corresponding to $X_{\sigma(k)}$ by applying Lemma 2.1.1, removing the picked jobs from the jobs available in the next window until we find the jobs available at time s_K. With the jobs available at time s_k, we can guess the optimal size of the next window by picking the slack time s_{K+1} from all slack times that are greater or equal to release time $r_{\sigma(K)+1}$.

With s_{K+1} we can guess the optimal choice for $X_{\sigma(K)} = \{x_t^{\sigma(K)}\}_{t\in P}$ from all possible

combinations of jobs with release time at least s_{K+1} and available at time s_K. First, the jobs picked greedily for $X_{\sigma(K)}$ are scheduled in increasing order of deadline right-shifted up to the slack time s_{K+1}, which can be done since every job has deadline at least s_{K+1}. We reject the choice for $X_{\sigma(K)}$ if there are jobs that start before s_K when scheduled in this way. Next, from the available jobs with deadline less than s_{K+1} we can assume they have release time s_K and deadline at most the start time of the jobs in $X_{\sigma(K)}$ and apply Moore's Algorithm [19] to them to get $Y_{\sigma(K)}$. We re-schedule these jobs to be in non-decreasing order of deadline by the Jackson Rule, so the jobs correspond to a canonical schedule.

It is left to check that our choice of jobs for $X_{\sigma(K)}$ and $Y_{\sigma(K)}$ are consistent with a canonical schedule. First, we check that every job in $X_{\sigma(K)}$ has start time in the half open interval $[r_{\sigma(K)}, r_{\sigma(K)+1})$ when scheduled in increasing order of deadline, breaking ties by scheduling in increasing order of job size, rejecting the current guess of s_{K+1} and $X_{\sigma(K)}$ if this does not hold. We also check that there is no idle time between $Y_{\sigma(K)}$ and $X_{\sigma(K)}$, and reject the current guess of s_{K+1} and $X_{\sigma(K)}$ if this does not hold. We reject in this case because we could left-shift $X_{\sigma(K)}$ and use an earlier slack time with no loss.

The dynamic program is indexed by collections of windows $\{[s_k, s_{k+1}]\}_{k=1}^{K}$ and the corresponding $X_{\sigma(k)} = \{x_t^{\sigma(k)}\}_{t \in P}$ choices, where K is the window to schedule in. For simplicity, we denote the pair of windows and jobs with a vector $\mathcal{X}(K) = \left\{([s_K, s_{K+1}], X_{\sigma(k)})\right\}_{k=1}^{K}$. The algorithm computes a solution for the table $A[\mathcal{X}(K)]$, which represents the maximum throughput in windows $\{[s_k, s_{k+1}]\}_{k=1}^{K}$ given the corresponding $X_{\sigma(k)} = \{x_t^{\sigma(k)}\}_{t \in P}$ choices. To check that a choice for $X(K)$ is feasible, we ensure that the choices have the properties of a canonical schedule. In increasing order of k, we find the jobs for $X_{\sigma(k)}$ greedily from the jobs with deadline at least s_{k+1}, and schedule them in non-decreasing order of deadline, breaking ties by scheduling in increasing order of job size, right-shifted to s_{k+1}. We check that every job schedule this way start in the interval $[r_{\sigma(k)}, r_{\sigma(k)+1})$, all jobs are wholly in the interval $[s_k, s_{k+1}]$. We also check that there are enough jobs for each $X_{\sigma(k)}$ guess, by ensuring that there are $x_t^{\sigma(k)}$ for each job type t.

We use these procedures to fill the table $A[\mathcal{X}(K)]$ using the following recursive formula

$$A[\mathcal{X}(K)] = \max_{s_{K+1}, X_{\sigma(K)}} [A[\mathcal{X}(K) + ([s_K, s_{K+1}], X_{\sigma(K)})] + |X_{\sigma(K)}| + |Y_{\sigma(K)}|]$$

Where s_{K+1} is picked from slack times $\mathcal{T}$ between $r_{\sigma(K)+1}$ and $r_{\sigma(K)+1} + p_{\max}$. The jobs in $X_{\sigma(K)}$ is an integer vector $\{x_t^{\sigma(K)}\}_{t \in P}$, and the jobs for it are picked greedily from the jobs available at time s_K with deadline greater than or equal to s_{K+1}. Finally, $Y_{\sigma(K)}$ are the jobs found by Moore's Algorithm [19].

2.3 Algorithm Analysis

Claim 2.3.1. *For any canonical schedule we can assume that the jobs $Y_{\sigma(K)}$ for each window $[s_K, s_{K+1}]$, can be chosen based on Moore's Algorithm*

Proof. For the jobs available at time s_K with deadline less than s_{K+1} we can view them as having their deadline as being no greater than the start time of the earliest job in $X_{\sigma(K)}$, since they cannot be feasibly scheduled with end time past this point. We can similarly view the start time of these jobs as s_K. Thus we have an instance of $1|r_j = 0|\sum_j U_j$, which can be solved by Moore's Algorithm [19]. Call the jobs chosen by Moore's Algorithm M.

Obviously, $|Y_{\sigma(K)}| \leq |M|$ since M is the optimal solution for this subproblem. Similarly, since $Y_{\sigma(K)}$ is part of the optimal solution for the whole problem we have we have $|Y_{\sigma(K)}| \geq |M|$. If instead we had $|Y_{\sigma(K)}| < |M|$, then we could replace $Y_{\sigma(K)}$ with M, and increase the value of the solution, which is a contradiction since we assumed $Y_{\sigma(K)}$ is optimal. Therefore, we can replace $Y_{\sigma(K)}$ with M and still have an optimal solution. □

Lemma 2.3.2. *We can fill the table A in polynomial time.*

Proof. The table A is indexed by windows and corresponding choices for X_i. Each window is a pair of slack times, which are defined as sums of job sizes with a release time. Therefore, there are no more than $O(n^{|P|})$ many slack times for a fixed release time, and so there are $O(|R|n^{|P|})$ many slack times. Each choice of X_i is an integer vector $\{x_t^i\}_{t \in P}$, where each x_t^i is in the range $[0, P]$. Therefore, there are $O(n^{|P|})$ possible choices for each X_i. Since there are $|R|$ many windows, there are $O(|R|n^{|P||R|})$ many choices for the vector $\mathcal{X}(K)$. Since $|P|$ and $|R|$ are assumed to be constant in size this value is polynomial in n.

To find the value of a table entry, we need to find the jobs for X_i, schedule those jobs, and then find the jobs for Y_i. Finding the jobs for X_i greedily is just taking the jobs in order of deadline which is clearly polynomial time. To schedule the jobs for X_i first sort them by deadline, then schedule them right shifted to the end of the window. Checking if these jobs satisfy the conditions of a canonical schedule is just checking that each job starts in the release interval and all jobs can be scheduled wholly within the window, which takes polynomial time.To find the schedule for Y_i we use Moore's algorithm, which takes polynomial time [19]. □

The above give the following theorem.

Theorem 2.3.3. *The dynamic programming algorithm finds an exact solution to $1|P \in O(1), R \in O(1)|\sum_j U_j$ in polynomial time.*

Chapter 3

Constant Number of Release Times and Deadlines

In this Chapter we give algorithms for the case of THROUGHPUT MAXIMIZATION where we assume there are a constant number of unique release times and deadlines, and a constant number of parallel machines. This case is written in the Graham notation as $Pm|\#r_j \in O(1), \#d_j \in O(1)|\sum_j U_j$. We first give an exact pseudo-polynomial time for this case with runtime that is polynomial in both n and $p_{\max}$, which we extend to find a PTAS. This gives the following

Theorem 3.0.1. *There is an algorithm that finds an exact solution to $Pm|\#r_j \in O(1), \#d_j \in O(1)|\sum_j U_j$ that is polynomial in both the number of jobs and $p_{\max}$*

Furthermore, if $p_{\max}$ is not bounded by a polynomial in the number of jobs, then this case admits a PTAS.

We will define the algorithm for the single machine case, and provide a simple extension to a constant number of machines at the end of the Chapter.

3.1 Problem Overview

Our approach will be to find *windows* where we can schedule jobs in much like in Chapter 2, but we restrict the number of deadlines instead of the number of job sizes so we make decisions for job types differently. We also change the definition of job types to make use of the constant number of release times and deadlines.

Definition 3.1.1 (Types)**.** *We say a job $j \in \mathcal{J}$ is of type $t = (u, v)$ if u is the release time of job j, r_j, and if v is the deadline of job j, d_j. We let T denote the set of all job types. And for type t, we let $\mathcal{J}_t$ denote the set of jobs belonging to type t.*

Since we assume the number of release times and deadlines is constant, we see that $|T| \in O(1)$. With these classifications, before scheduling individual jobs, we first guess how

much processing time each type t has in an optimal solution and use this guess as a budget for job processing times and maximize the number of jobs of type t scheduled given this budget.

We call the union of release times R and deadlines D *straddle points*, which we denote as $\mathcal{S}$. We enumerate the points in $\mathcal{S}$ so that $s_i \in \mathcal{S}$ is the ith point in increasing order. We will explicitly guess which jobs are straddling each straddle point in an optimal solution, and schedule the remaining jobs in the unused intervals between straddle points. We can represent the choice of such a job, called a *straddle job*, by the triple $(j, s_i, s_k) \in (\mathcal{J} + \perp) \times \mathcal{S}^2$, where j is the job in question that straddles only the points $\{s_{i+1}, \ldots, s_k\}$, and $s_i < s_k$. The symbol $\perp$ denotes that there is no job straddling the points in question, and may only be in the triple $(\perp, s_i, s_{i+1})$, meaning no job straddles the point s_{i+1}, we still refer to this case as being a *straddle job*. The choice of straddle jobs is represented with the vector $\vec{S}$.

Definition 3.1.2. *We say that a choice of straddle jobs is* regular *if;* (1) *each job is in $\vec{S}$ at most once,* (2) *for $(j, s_i, s_k) \in \vec{S}$, if $j \neq \perp$ then $r_j \leq s_i$ and $s_k \leq d_j$, and $p_j \geq |s_i - s_k|$,* (3) *every straddle point is straddled by exactly one straddle job, except s_1 and $s_{\max}$ which are not straddled.*

If $\vec{S}$ is regular then we can define *windows*, which will denote the intervals where we schedule non-straddle jobs. Given straddle jobs (j, s_i, s_k) and $(j', s_k, s_{k'})$, window k is the interval between jobs j and j'. Also window 1 is between s_1 and (j, s_1, s_k), and window $\max -1$ is between $(j, s_k, s_{\max -1})$ and $s_{\max}$. Note that we do not have the exact size of the windows since we do not specify the start times of straddle jobs. We define windows in this way so that we can use the set of windows, denoted $\mathcal{W}$ to index the next definition.

To find window size we also need to know how much processing time is used by jobs in the window. For window $i \in \mathcal{W}$, the total processing time of jobs of type t scheduled in this window is the *allotment* of that job type, and is denoted $a_{i,t}$. We denote the allotments of the jobs in the window by $\vec{a}_i = \{a_{i,t}\}_{t \in T}$, and the allotments for the whole schedule by $\vec{a} = \{\vec{a}_i\}_{i \in \mathcal{W}}$. The size of window i is therefore at least as large as the sum of allotments in $\vec{a}_i$, but it could be larger in that case the there is idle time between straddle jobs.

To provide intuition for our algorithm we will assume for the moment that there are no straddle jobs, that is $\vec{S} = \{\perp, s_i, s_{i+1}\}_{i=2}^{\max -2}$, therefore there is a window for every consecutive pair of straddle points. Suppose this case has optimal solution $\mathcal{O}$. In a left to right manner, for each window i, we guess the allotment of $\mathcal{O}$, $\vec{a}_i$, assuming that the sum of the allotments does not exceed the space between the two straddle points. Since the jobs in a window can be reordered and scheduled in any order we can say that the jobs of a fixed type would be scheduled in a contiguous block. Therefore, we can treat allotments as contiguous blocks where we schedule jobs of a fixed type. Given these allotments, we schedule the jobs of each type independently using Multiple Knapsack methods described in appendix A.

3.2 Pseudo-Polynomial Time Algorithm

To find an pseudo-polynomial time exact solution to the problem we will find a schedule by guessing the straddle jobs and allotments in an optimal solution and then scheduling jobs into their allotments by focusing on jobs of a fixed type. This approach is only pseudo-polynomial since its runtime is polynomial in the maximum job size, $p_{\max}$, as well as n. To see that this approach finds an optimal schedule, we will take an optimal schedule $\mathcal{O}$, and reschedule its jobs to nicely adhere to the definitions of straddle jobs and allotments. This will then define *canonical* schedules that we can enumerate over in our algorithm to find an optimal solution.

First left-shift $\mathcal{O}$ and denote the jobs straddling the straddle points by $\vec{S}^*$, a vector of triples $(j, s_i, s_k) \in (\mathcal{J} + \bot) \times \mathcal{S}^2$. For every pair of triples $(j, s_{i'}, s_i)$ and (j', s_i, s_k) in $\vec{S}^*$, between consecutive straddle jobs, there is a *window* between jobs j and j'. The set of windows induced by the the straddle jobs $\vec{S}^*$ is denoted $\mathcal{W}^*$.

For window every i in $\mathcal{W}^*$, since the schedule is left shifted, the jobs between two consecutive straddle jobs start at the end time of a straddle job and are entirely between two straddle points. Therefore, we can feasibly schedule these jobs in any order, in particular we can schedule them in groups according to type, left-shifted to the nearest straddle job. The size of these groups are the allotments, which we denote as $\vec{a}_i^*$ for window i, and $\vec{a}^* = \{\vec{a}_i^*\}_{i \in \mathcal{W}}$ is the vector of all allotments.

Observation 3.2.1. *Every job in $\mathcal{O}$ is either a straddle job or scheduled in an allotment.*

This observation follows from the definition of $\vec{S}^*$ and $\vec{a}^*$, since either a job is a straddle job, or is used to define an allotment.

Observation 3.2.2. *For every straddle job j in $\vec{S}^*$, the start time of j is either; the end point of an allotment, or the end point of a straddle job.*

Similarly, for every allotment in $\vec{a}^*$, if it is the first allotment scheduled in a window, then its start point is the endpoint of a straddle job, otherwise its start point is the end point of another allotment.

This observation follows by the way we define straddle jobs in $\vec{S}^*$, where if no job straddles a point s_i then we say that there is a straddle job $(\bot, s_i, s_{i+1})$. Combined with left-shifted property of $\mathcal{O}$, since left-shifting moves jobs left up to either their release time or the end time of another job, we will left-shift jobs to the endpoint of a straddle job.

Observation 3.2.3. *$\vec{S}^*$ is a regular choice of straddle jobs.*

We say that schedules that are organized into straddle jobs and allotments are *canonical schedules*, and we provide an example in Figure 3.1.

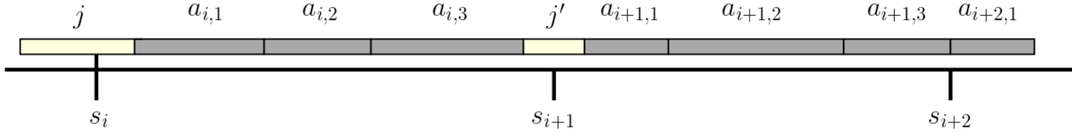

Figure 3.1: In this example the grey blocks represent allotments and the white blocks represent the straddle jobs of a canonical schedule.

Lastly, we have the following observation that will be important for finding optimal canonical schedules.

Observation 3.2.4. *Given the allotments $\vec{a}^*$ and straddle jobs $\vec{S}^*$ for $\mathcal{O}$, the problem of scheduling jobs of type t is independent of every other job type.*

This last observation is important as it allows our algorithm to deal with each job type independently. This is clearly true since each job type has a specified allotment that jobs of that type can be scheduled in, and the allotments of two job types do not overlap.

3.2.1 Algorithm

The algorithm here is a sweep across all canonical schedules by iterating straddle jobs and allotments, combined with a Multiple Knapsack dynamic program to schedule jobs of each type in their corresponding allotments.

Our algorithm begins by iterating across all vectors of straddle jobs $\vec{S}$, with entries (j, s_i, s_k), requiring that $\vec{S}$ are regular straddle jobs. Given a regular choice of straddle jobs, between each consecutive pair of straddle jobs in increasing order, we will guess an optimal choice of allotments, $\vec{a}$, where $a_{i,t} \in [0, np_{\max}]$. We check that this choice of allotments corresponds to a canonical schedule by checking if the allotments can be scheduled feasibly as if they were jobs. This allotment placement will simultaneously define the windows of the schedule and will be used when scheduling the jobs. We let window 1 begin from point s_1 and check that the point $s_1 + \sum_{t=1}^{T} a_{1,t}$ is at most the next straddle point s_2, if not then the check fails as the allotments are too large to fit in the window. If this check succeeds, then take straddle job (j, s_1, s_k) and if $j \neq \perp$ then schedule it left-shifted to the allotments, checking that it still ends after s_k, if not then the endpoint for j should have been in an earlier interval, so we reject the allotment guess. We then repeat this process from the endpoint of job j, finding a schedule for the allotments and straddle jobs. We also check that for any $a_{i,t} \neq 0$, that the release time of type t is at most s_i, and the deadline of type t is at least s_{i+1}, this ensures that when the jobs are scheduled in their allotments they are scheduled feasibly.

With a choice of straddle jobs and allotments that correspond to a canonical schedule, we place the allotments and straddle jobs as described in the check for allotments. With this placement, we apply Observation 3.2.4 to reduce the problem to solving an instance of the Multiple Knapsack problem for each job type. For the problem corresponding to jobs of type t, say there is a knapsack m_i corresponding to every window i, of size $a_{i,t}$, and for each job j of type t there is a corresponding item, x_j in the Multiple Knapsack problem, with weight equal to p_j and profit of 1. We know by Appendix A that instances of the Multiple Knapsack problem with $O(1)$ many knapsacks can be solved exactly in pseudo-polynomial time. To revert a solution of the Multiple Knapsack problem corresponding to jobs of type t to a schedule, we look at each knapsack m_i and the items packed in it. If x_j is packed into knapsack m_i, then schedule job j in allotment $a_{i,t}$.

3.2.2 Algorithm Analysis

Claim 3.2.1. *This algorithm runs in pseudo-polynomial time*

Proof. We first check that the number of possible straddle jobs that we guess is bounded by a polynomial in n. Since there are n jobs, and $O(R+D) = O(1)$ many straddle points there are $O(n^{R+D}) = n^{O(1)}$ many possible ways to assign jobs. Each straddle job is also assigned a pair of intervals that it must start and end in, so for a particular job j the number of pairs assigned to j is upper bounded by $O((R+D)^2)$. If we fix a job choice for each straddle point, since there are $O(R+D)$ many straddle points the number of possible choices of $\vec{S}$ is upper bounded by $O((R+D)^{(R+D)^2})$, which is a constant value since both R and D are constant. Therefore, the number of choices for $\vec{S}$ is bounded by $n^{O(1)} \cdot O((R+D)^{(R+D)^2}) = n^{O(1)}$.

Next we show that the number of choices for allotments is pseudo-polynomial. For each job type, we guess at most $O(R+D)$ many allotments, each with a size in $[0, np_{\max}]$. There are $R \cdot D \in O(1)$ many types so there are $O(np_{\max})$ many choices for allotments. Placing these allotments to create the windows takes a polynomial time since it is simply scheduling the allotments as if they were jobs and left shifting them.

The reduction to the Multiple Knapsack problem is clearly polynomial since we create one item for each job with weight equal to the job size and profit one, and we create knapsack for each allotment. Similarly, reverting the solution to the Multiple Knapsack problem takes polynomial time.

Lastly, we know by appendix A that the Multiple Knapsack problem with uniform profits can be solved in pseudo-polynomial time. □

Claim 3.2.2. *This algorithm gives an exact solution to* Throughput Maximization *with a constant number of release times and deadlines.*

Proof. We know we accept canonical choices of straddle jobs and allotments, and thus we

will pick $\vec{S}^*$ and $\vec{a}^*$ of the optimal schedule $\mathcal{O}$. When we check that these choices are optimal we also schedule them in a left-shifted manner as in the canonical schedule $\mathcal{O}$, so by Observation 3.2.4 which says we can schedule each job type independent of the others given $\vec{S}^*$ and $\vec{a}^*$, we need to see that we find an optimal schedule for each job type. From appendix A we know that the MULTIPLE KNAPSACK problem can be solved optimally via dynamic programming. □

Theorem 3.2.3. *There is an algorithm that finds an exact solution to* $1|R \in O(1), D \in O(1)|\sum_j U_j$ *that is polynomial time in the number of jobs.*

3.3 PTAS for Constant Number of Release Times and Deadlines

If job sizes are not assumed to be bounded by a polynomial in n then the runtime of our algorithm has two problems. The first, is that we make $O(np_{\max})$ many guesses for each allotment. Second, we exactly solve the MULTIPLE KNAPSACK problem using an algorithm with runtime that is polynomial with respect to both n and $p_{\max}$. In both cases, the dependence of the runtime on a polynomial in $p_{\max}$ keeps the algorithm from having a runtime that is polynomial in the input size. To deal with the second problem, we use the PTAS from [6] to find a schedule.

Given an optimal canonical schedule $\mathcal{O}$, for each allotment $a_{i,t}$, if the allotment has at least $\lceil 1/\varepsilon^2 \rceil$ jobs then we reduce the size of the allotment to the nearest power of $(1+\varepsilon)$ and and drop jobs in order from largest to smallest until the remaining jobs can be scheduled entirely in this reduced allotment.

Lemma 3.3.1. *Given optimal canonical schedule* $\mathcal{O}$*, if we apply the above rounding procedure then the throughput of this new schedule is a* $(1-2\varepsilon)$*-approximation of the throughput of* $\mathcal{O}$*.*

Proof. Take an optimal canonical schedule $\mathcal{O}$, for a fixed window, if an allotment has at least $B = \lceil 1/\varepsilon^2 \rceil$ jobs then we round down the size of the allotment to the nearest power of $(1+\varepsilon)$. We drop jobs in order of largest to smallest until the remaining jobs fit in the allotment.

We want to show that the fraction of jobs remaining after this rounding is at least $\frac{1}{1+\varepsilon}$. The worst case for this fraction is when the jobs in this allotment is exactly B many jobs. Rounding the allotment size down to the nearest $(1+\varepsilon)$ power means that there will be at least $\lfloor \frac{B}{1+\varepsilon} \rfloor$ jobs. If we let $B = \frac{1}{\varepsilon^2}$, the fraction of jobs remaining will be

$$\frac{\lfloor \frac{B}{1+\varepsilon} \rfloor}{B} \geq \frac{\frac{B}{1+\varepsilon} - 1}{B} = \frac{1}{1+\varepsilon} - \frac{1}{B} = \frac{1}{1+\varepsilon} - \varepsilon^2 = \frac{1-\varepsilon^2-\varepsilon^3}{1+\varepsilon}$$

Rearranging terms of $\frac{1-\varepsilon^2-\varepsilon^3}{1+\varepsilon}$ we see the following

$$\frac{1-\varepsilon^2-\varepsilon^3}{1+\varepsilon} = 1-(1-\frac{1-\varepsilon^2-\varepsilon^3}{1+\varepsilon}) = 1-\varepsilon\frac{1+\varepsilon+\varepsilon^2}{1+\varepsilon} \geq 1-2\varepsilon$$

Where the last inequality follows from the fact that $\frac{1+\varepsilon+\varepsilon^2}{1+\varepsilon} = 1+\frac{\varepsilon^2}{1+\varepsilon} \leq 1+\varepsilon$

$$\Rightarrow \frac{\lfloor \frac{B}{1+\varepsilon} \rfloor}{B} \geq 1-\varepsilon' \geq 1-2\varepsilon$$

□

3.3.1 Algorithm

The algorithm we use will be similar to the exact algorithm. We will sweep across straddle jobs as before, checking that they correspond to canonical schedules. To sweep across allotments, we will guess from both allotment sizes that are powers of $(1+\varepsilon)$ and that are equal to combinations of up to $\lceil 1/\varepsilon^2 \rceil$ many job sizes. The reduction to the Multiple Knapsack problem is the same but instead of the pseudo-polynomial time solution, we use the PTAS due to [6].

3.3.2 Analysis

Claim 3.3.2. *This algorithm runs in polynomial time.*

Proof. By restricting the number of guesses for allotments to powers of $(1+\varepsilon)$, we bring the number of guesses for allotments from polynomial in the $p_{\max}$ to being polynomial in the size of the input. The size of jobs does not affect the choice of straddle jobs since we schedule them by greedily filling the timeline with allotments and straddle jobs in a left shifted manner. Thus the number of guesses for straddle jobs remains bounded by $n^{R+D} = n^{O(1)}$.

Our reduction to the Multiple Knapsack problem maps jobs to items, and allotments to knapsacks. We use the PTAS due to [6] to find a $(1-O(\varepsilon)$-approximate solution to the Multiple Knapsack problem, which has runtime that is polynomial in the number of items, n, and the number of knapsacks, a constant. This solution to the Multiple Knapsack problem is then transformed back to a schedule for the original problem by mapping each item in a knapsack to the corresponding job and allotment. □

Theorem 3.3.3. *This algorithm is a PTAS for the* Throughput Maximization *problem with a constant number of release times and deadlines.*

Proof. We know that we restrict our choice of allotments in such a way that there is optimal schedule is within $(1-2\varepsilon)$ of the original optimal. Given this choice of allotments, we need to find a solution to a constant number of Multiple Knapsack problems with identically

weighted jobs of arbitrary size which we can solve using a PTAS due to [6]. Therefore, we find a solution that is at least a $(1-2\varepsilon)(1-\varepsilon) = 1-O(\varepsilon)$ factor of the optimal solution. □

3.4 General Case is Weakly NP-Complete

Theorem 3.4.1. *The* Throughput Maximization *problem with a constant number of release times and deadlines is Weakly NP-Complete*

Proof. First, we show the problem is in NP. We are given an instance T of Throughput Maximization with a constant number of release times and deadlines, a goal k for the number of jobs throughput. For a witness schedule S, a simple verifier algorithm for S is to take each of the n jobs, and schedule them starting at the time specified in S, then check that the processing time of job is disjoint from every other job, which can be done in $O(n^2)$ time. Lastly, the algorithm returns Yes if at least k jobs are scheduled entirely within their release time and deadline, which takes $O(n)$ time. Thus the problem is in the class NP.

Next, we show that the Partition problem can be reduced to the Throughput Maximization problem, thus completing the proof.

Take an instance of the Partition problem $P = \{p_1, \ldots, p_n\}$. Our reduction is as follows; for each elements p_i we create a job j_i of processing time p_i with release time $r_1 = 0$ and deadline $d_1 = \sum_{i=1}^{n} p_i + 1$. We also add job j' with release time $r_2 = \frac{1}{2}\sum_{i=1}^{n} p_i$ of size 1 with deadline $d_2 = r_2 + 1$.

We claim that there is a schedule with throughput $n+1$ if and only if we can solve the Partition problem.

If we can find a scheduled with throughput $n+1$, then we can find a partition by letting $S_1 \subset P$ be the elements corresponding to the jobs scheduled in the interval (r_1, r_2), and $S_2 \subset P$ be the elements corresponding to the jobs scheduled in the interval (d_2, d_1). Since the throughput is $n+1$, the jobs j' is scheduled at r_2, and thus partitions the timeline into (r_1, r_2), and (d_2, d_1). By definition we have $|r_1 - r_2| = |d_1 - d_2| = \frac{|P|}{2}$, so because all jobs are scheduled the entire interval is tightly packed since the job sizes sum up to $|P|+1 = d_1$. Therefore, the jobs in the interval (r_1, r_2) have total processing time $\frac{|P|}{2}$ so when we pick the elements S_1 for the partition problem the elements sum to $\frac{|P|}{2}$. A similar argument applies to S_2 for the Partition problem, and so $|S_1| = |S_2|$.

Suppose we can find a solution to the Partition instance, $S_1 \cap S_2 = P$. This is then a solution with throughput $n+1$ for the scheduling problem. □

3.5 Extending to a Constant Number of Machines

In this Section we describe how to extend the results of this Chapter to a constant number of machines to get an exact pseudo-polynomial time solution to $Pm|\#r_j \in O(1), \#d_j \in$

$O(1)|\sum_j U_j$. The intuition of this extension is clear, as before we guess optimal straddle jobs for each straddle point and guess optimal allotments between straddle jobs, except we guess one optimal straddle job for each machine at each straddle point, and guess optimal allotments for each machine. Straddle jobs are tuples $(i, j, s_k, s_l) \in \{1, \ldots, m\} \times (\mathcal{J} + \perp) \times \mathcal{S}^2$ where i is the machine the job is scheduled in, j is the job in question, straddling only the points $\{s_{i+1}, \ldots, s_k\}$, *regular* straddle jobs are defined in much the same way except point (3) says that each straddle point for each machine is straddled exactly once. Windows are defined similar to the single machine case, on a machine by machine basis, look at the straddle jobs and define the windows for that machine. Similarly, the allotments are defined on a machine by machine basis given the windows.

Since there are now $O(m(R + D))$ many possible straddle points we need to consider when picking $\vec{S}$, the number of possible choices for $\vec{S}$ is upper bounded by $n^{O(1)} \cdot O((m(R + D)^{m(R+D)^2} = n^{O(1)}$. Similarly, the number of windows increases by at most a factor of m so the number of possible allotment guesses is bounded by $O(mnp_{\max}) = O(np_{\max})$.

With multiple machines we can define *canonical* schedules in a similar way as the single machine case. Take a left-shifted optimal solution $\mathcal{O}$, and fix the jobs straddling the straddle points, creating straddle job $(i, \perp, s_k, s_{k+1})$ if there is no job straddling point s_{k+1} on machine i. For each machine, between two consecutive straddle jobs we can sort the jobs by job type, and make Observations equivalent to Observations 3.2.1, 3.2.2, and 3.2.3 for each machine. We can also make an observation equivalent to Observation 3.2.4, which say that jobs of each type can be scheduled in their allotments independently of all other job types.

The algorithm is a straightforward extension of the algorithm for single machines. We guess the optimal choice of straddle jobs $\vec{S}$ and allotments $\vec{a}$. To check these choices correspond to a canonical schedule, we perform the check described in Subsection 3.2.1 on a machine by machine basis. To find the schedule given these allotments, we perform the same reduction to Multiple Knapsack problems and use the algorithm described in Appendix A. Since the number of knapsacks increases by a factor of at most m, the algorithm still runs in time polynomial in n and $p_{\max}$.

We also have that Lemma 3.3.1 holds for this problem since it argues on a per allotment basis. So we can get a PTAS for for this problem by guessing optimal straddle jobs as before, and picking allotments that are either powers or $(1+\varepsilon)$ or are equal to combinations of up to $\rceil 1/\varepsilon^2 \lceil$ many job sizes. We reduce to the Multiple Knapsack problem as before and again apply the PTAS due to [6], noting that since the number of allotments increase by a factor of at most m, the algorithm of [6] still runs in polynomial time.

Chapter 4

Constant Number of Release Times

In this Chapter, our main result is an exact pseudo-polynomial time algorithm for the case of Throughput Maximization where the number of release times between the release time and deadline of any job is at most a constant, on a constant number of machines. This case can be written in Graham notation as $Pm|\#size_j \in O(1)|\sum_j U_j$. This gives the following theorem

Theorem 4.0.1. *There is an algorithm that finds an solution to* $Pm|span_j \in O(1)|\sum_j U_j$ *and runs in time polynomial in* n *and* $p_{\max}$*.*

To create this algorithm, we will use as a sub-routine an algorithm for the more restricted case of Throughput Maximization where there are at most a constant number of unique release times which we find in Section 4.1. In both cases, the algorithms will be defined in the single machine case and extended to a constant number of machines at the end of the Chatper.

4.1 Constant number of Release Times

In this section we examine Throughput Maximization where there are at most a constant number of release times, which is written as $1|\#r_j \in O(1)|\sum_j U_j$ in the Graham notation. We will prove the following theorem by providing an exact solution to this problem.

Theorem 4.1.1. *There is an algorithm that finds an exact solution to* $1|\#r_j \in O(1)|\sum_j U_j$ *that runs in pseudo-polynomial time.*

This result generalizes the results of Chapter 3 by relaxing the restriction on the number of unique deadlines. Our algorithm for Theorem 4.1.1 follows a similar approach to the algorithm in Chapter 3, in that we will guess straddle jobs and allotments for job types and use these guesses to schedule jobs based on type. However, since we do not assume

a constant number of deadlines we do not have a constant number of job types and so we can't efficiently run the algorithm of Chapter 3. To account for this, we will change the definitions from the previous sections slightly to be more amenable to the approach we will use here.

First we change the definition of job type to be a release time and release interval pair, where the deadline of the job is in the release interval.

Definition 4.1.1 (Job Type). *We say a job $j \in \mathcal{J}$ is of type $t = (s, [r_i, r_{i+1}))$ if $r_j = s$ and $d_j \in [r_i, r_{i+1})$. We call the interval $[r_i, r_{i+1})$ the* deadline interval *for the type. We let T denote the set of all job types.*

We also change the definition of *straddle points* to only refer to unique release times, since the number of deadlines is not bounded. So for a given schedule, a window is an interval between jobs straddling two consecutive straddle points, and therefore may contain deadlines. We use a similar definition of *straddle jobs*, a vector that describes the job scheduling over each release time. However, since the runtime of our algorithm is polynomial in $p_{\max}$, we can explicitly decide where straddle jobs end in a schedule by defining straddle jobs as a vector $\vec{S} \in ((\mathcal{J} + \perp) \times [0, p_{\max} - 1])^R$, a vector of job/integer pairs (j, s), where the ith entry of $\vec{S}$ is (j_i, s_i) and j_i straddles r_i, scheduled with completion time $r_i + s_i$. We have $s_i = 0$ if and only if $j_i = \perp$, which represents no straddle job, and thus it ends at time r_i. Given straddle jobs $\vec{S}$ we can define *windows* as the interval between two consecutive straddle jobs, so if the job straddle (j_i, s_i) has completion time $r_i + s_i$ in release interval i, then the window i is the interval between the completion time of j_i and the start time j_{i+1} from (j_{i+1}, s_{i+1}).

For this problem the definition of allotments is mostly the same as in Chapter 3, in that it describes intervals where jobs of fixed types will be scheduled. An *allotment* is an integer matrix $\vec{a} = \{\vec{a}_i\}_{i=1}^R$, where $\vec{a}_i = \{a_{i,t}\}_{t \in T}$. We understand $a_{i,t}$ as the total processing time for jobs of type t in window i except when window i is the deadline window for jobs of type t, in which case $a_{i,t} = 0$. For clarity, we say a job is scheduled "in an allotment" if it is one of the jobs that make up the size of the allotment. This is because jobs cannot be scheduled in arbitrary order in their deadline window like when they are scheduled in allotments so we do not have an allotment there. This means that jobs are not necessarily scheduled in their allotments and will be scheduled in a particular manner in their deadline window.

The intuition for our algorithm is similar to the algorithm for Chapter 3, but in this case since jobs might not be scheduled in one of their allotments there is added complexity. For ease of exposition, we explain the algorithm assuming that there are no straddle jobs, as adding them only changes the window sizes and not how we create job schedules.

Given a guess for allotments, for every release interval we schedule the allotments for the window in contiguous blocks as if they were jobs in increasing order of deadline interval

in a right-shifted manner up to the end point of the interval. For each release interval i, we will find a schedule for all job types with deadline interval i, denoted $\mathcal{J}_i$ with a dynamic program. We sort the jobs in $\mathcal{J}_i$ in non-decreasing order of deadline and enumerate them as $\mathcal{J}_i = \{j_1, \ldots, j_{|\mathcal{J}_i|}\}$.

The algorithm is a dynamic program that schedules a subset of $\mathcal{J}_i$ by considering every job in order of least deadline to greatest one at a time, and deciding where the job currently considered should be scheduled. If j_q is the job and is scheduled in an allotment, then the current allotment vector has a single value decremented by p_{j_q}, assuming the resulting value is non-negative and is an allotment for the same job type as j_q. If j_q is not scheduled at all, then simply ignore it and schedule j_{q+1}. Finally, there is the case if j_q is scheduled in its deadline window, where j_q is scheduled right-shifted in the deadline interval up to at most the allotments, assuming it starts no sooner than r_i. So j_q will end at either the start time of the allotments, another jobs start time, or its own deadline d_{j_q},

This procedure is repeated for every release interval to generate the whole schedule. The algorithm then finds the best schedule by iterating across all choices of allotments that are reasonably similar to an optimal schedule that we will define in subsection 4.1.1. We will deal with straddle jobs in a similar manner as Chapter 3, by iterating over reasonable choices of straddle job vectors in order to find a choice that is in an optimal schedule.

4.1.1 Canonical schedule

We will show that there exists an optimal schedule that possesses a useful structure that we call a *canonical schedule*, which we do by taking an optimal schedule $\mathcal{O}$ and rescheduling the jobs in it to be in this canonical form.

First, we fix the jobs straddling the straddle points in place and let them be represented by the vector $\vec{S}^*$. For every release time r_i if job j is straddling r_i with completion time $r_i + s_i$ then (j, s_i) is added to $\vec{S}^*$, adding $(\perp, 0)$ if there is no job straddling r_i. With $\vec{S}^*$, we can define the windows as the intervals between two consecutive straddle jobs, enumerated by the release interval they are contained in.

For each window in $\mathcal{O}$, we apply the Jackson Rule to reschedule the jobs in increasing order of deadline. For window i, jobs with release time earlier than r_i and deadline interval later than release interval i can obviously be rescheduled in any order within the window. Therefore, for each deadline interval we can sort jobs that share this deadline interval by type, scheduling the jobs of each type in increasing order of deadline. This gives us the following observation, which is demonstrated in Figure 4.1.

Observation 4.1.1. *For each window i in schedule $\mathcal{O}$, the jobs can be right-shifted, and those jobs with deadline after window i can be scheduled in groups according to job type after jobs with deadline interval i.*

For each job type t, the allotment for jobs of type t is exactly the sum of the processing time of this type, which we denote $\vec{a}^*_{i,t}$. The vector of all allotments in a window is $\vec{a}^*_i$, and the vector for all allotments is then $\vec{a}^*$. We note that the jobs scheduled in each allotment are scheduled in increasing order of deadline.

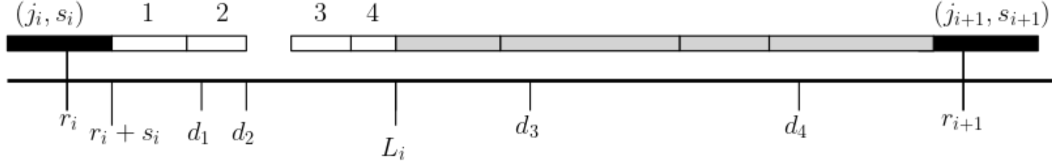

Figure 4.1: In this example, the dark blocks are the straddle jobs (j_i, s_i) and (j_{i+1}, s_{i+1}), and the light grey blocks are the allotments. The jobs $\{1, 2, 3, 4\}$ are the jobs with deadline in this interval and are scheduled before the allotments in increasing order of deadline and right-shifted to the starting time of the allotments, which is denoted L_i.

We say that any schedule satisfying Observation 4.1.1 is a *canonical* schedule. Given straddle jobs $\vec{S}^*$ and allotments $\vec{a}^*$, the schedule for jobs with deadline interval i can be found independently of all other jobs. This follows since jobs with deadline interval i are either scheduled in allotments or in the first part of the deadline interval that is dedicated to jobs with deadline interval i.

4.1.2 Algorithm

We first guess a vector of optimal straddle jobs, $\vec{S} \in ((\mathcal{J} + \perp) \times [0, p_{\max} - 1])^R$. We ensure that $\vec{S}$ corresponds to some canonical schedule by checking that for every pair $(j_i, s_i) \in \vec{S}$, j_i can be scheduled with completion time $r_i + s_i$ while straddling release time r_i, and that $r_i + s_i \leq d_{j_i}$. We also check that if a job is part of multiple pairs in $\vec{S}$, that the end time given in each pair is the same. We lastly check that when straddle jobs are scheduled following these rules, that no straddle jobs are processing at the same time.

We next guess allotments $\vec{a} = \{\vec{a}_i\}_{i=1}^R$ in an optimal schedule, where $a_{i,t} \in [0, np_{\max}]$. For type $t = (r_k, [r_{k'}, r_{k'+1}])$ we also require that $a_{i,t} = 0$ for all i less than k, and at least k'. That is, we require that non-zero allotments only come after the release time and before the deadline window of the job type. With a guess for $\vec{S}$ that corresponds to a canonical schedule, we check that the allotments also correspond to a canonical schedule. For each window i, we check if the sum of allotments for this window $\sum_{t \in T} a_{i,t}$, is less than the length of the interval $[r_i + s_i, r_{i+1} + s_{i+1} - p_{j_{i+1}}]$, which is the interval between the two consecutive straddle jobs.

We denote the set of jobs with deadline in release interval i by $\mathcal{J}_i$, and the jobs of $\mathcal{J}_i$ not picked to be straddle jobs are denoted by $\mathcal{J}_i(\vec{S})$. With a guess for straddle jobs and allotments corresponding to a canonical schedule, we define dynamic program to find an

optimal schedule for $\mathcal{J}_i(\vec{S})$. We first sort and enumerate the jobs in $\mathcal{J}_i(\vec{S})$ in increasing order of deadline and enumerate them so $\mathcal{J}_i(\vec{S}) = \{j_1, \ldots, j_{|\mathcal{J}_i(\vec{S})|}\}$.

The recursive formula for our algorithm iterates through the jobs of $\mathcal{J}_i(\vec{S})$ in order of smallest deadline to largest to decide how they are scheduled. The job with latest deadline will either be scheduled in an allotment right-shifted, or in its deadline window right-shifted to at most the allotments, or the job is not scheduled at all.

Given a choice of straddle jobs and allotments, once they have been scheduled for each window i, there is an interval between the end time of the straddle job and the start time of the allotments, which we call L_i. This interval is where we will schedule jobs with deadline interval i, and we will track the size of this interval in our dynamic program. To keep the size of this interval from being larger than $np_{\max}$, the largest amount of processing time for all jobs, we assume that this interval is instead between r_i+s_i and $L := \min\{r_i+s_i+np_{\max}, L_i\}$.

For a guess for optimal straddle jobs, $\vec{S}$, and allotments, $\vec{a}$, our algorithm computes the table $A_i[\vec{a}, j_{|\mathcal{J}_i(\vec{S})|}, L]$, which is the maximum throughput of the jobs in $\mathcal{J}_i(\vec{S})$, which are scheduled in both the allotments of types with deadline in release interval i, and between the completion time of the straddle job of r_i and the time L.

To compute $A_i[\vec{b}, j_q, l]$ with $q > 1$, j_q is the job with least deadline that has not been examined, we must decide how to schedule it in a canonical schedule. The jobs will be scheduled in either $\vec{b}$, the current allotment sizes, or in $[r_i + s_i, l]$, the remaining unused space in the deadline interval. If j_q is not in the optimal schedule, then the optimal schedule is composed of jobs $\{j_1, \ldots, j_{q-1}\}$, so we know that

$$A_i\left[\vec{b}, j_q, l\right] = A_i\left[\vec{b}, j_{q-1}, l\right]$$

If j_q is in the optimal schedule, then it is either scheduled in an allotment, or in its deadline interval. If it is scheduled in release interval i, the deadline interval of $\mathcal{J}_i(\vec{S})$, in the optimal schedule, then we schedule it right-shifted to at most l, but no further than d_{j_q}. So the space for jobs in the deadline interval decreases to $\min\{d_{j_q}, l - p_{j_q}\}$. This means that

$$A_i\left[\vec{b}, j_q, l\right] = A_i\left[\vec{b}, j_{q-1}, \min\{d_{j_q}, l - p_{j_q}\}\right] + 1$$

If j_q is instead scheduled in some allotment $b_{k,t}$ in an optimal schedule, where t is the job type of j_q, then we can can schedule it right-shifted in the allotment $b_{k,t}$. This means we reduce the size of $b_{k,t}$ by p_{j_q} to get $\vec{b}'$, where $\vec{b}'$ is equal to $\vec{b}$ except the allotment job j_q is scheduled in, which is $b_{k,t} - p_{j_q}$ instead of $b_{k,t}$, ensuring that $b_{k,t} - p_{j_q}$ is non-negative. The set of allotments that satisfy this condition is $B(q,t)' = \{b' | \exists k, b_{k,t} - p_{j_q} \geq 0\}$. This means

$$A_i\left[\vec{b}, j_q, l\right] = \max_{\vec{b}' \in B(q,t)} \left\{A_i\left[\vec{b}', j_{q-1}, l\right] + 1\right\}$$

Therefore, if we have computed the values of A_i with j_{q-1}, we know the optimal schedule can be found by the following recursive formula.

$$A_i\left[\vec{b}, j_q, l\right] = \max \begin{cases} A_i\left[\vec{b}, j_{q-1}, l\right] \\ A_i\left[\vec{b}, j_{q-1}, \min\{d_{j_q}, l\} - p_q\right] + 1 \\ \max_{\vec{b'} \in B(q,t)} \left\{ A_i\left[\vec{b'}, j_{q-1}, l\right] + 1 \right\} \end{cases}$$

If $q = 1$, then obviously the optimal schedule is if j_1 can be scheduled anywhere. Therefore, by induction the following Lemma holds

Lemma 4.1.2. *Suppose we have guessed the straddle jobs $\vec{S}^*$ and allotments $\vec{a}^*$ of an optimal canonical solution $\mathcal{O}$. Then, the dynamic program finds an optimal canonical schedule for each table A_i.*

Given an initial guess for straddle jobs and allotments, we compute tables A_i for each window i. We finally check that this schedule is a feasible one by ensuring that every job j that is scheduled in the range $[r_j, d_j]$. Our algorithm returns the maximum sum of these tables across all accepted straddle jobs and allotments.

4.1.3 Analysis

Lemma 4.1.3. *The algorithm finds an optimal canonical schedule.*

Proof. By Lemma 4.1.2 we know that if we guess the straddle jobs and allotments from a canonical schedule, our algorithm finds an optimal canonical schedule for each table A_i, and therefore, when run for each release interval finds an optimal canonical schedule. Recall from Subsection 4.1.1 that there is an optimal canonical schedule $\mathcal{O}$ with vectors of straddle jobs $\vec{S}^*$ and allotments $\vec{a}^*$. To see that the algorithm finds an optimal canonical schedule, we want to see that we eventually choose both $\vec{S}^*$ and $\vec{a}^*$.

To see that $\vec{S}^*$ is eventually chosen, we just need to see that it does not fail the checks we impose on it. The first check is that every pair $(j_i, s_i) \in \vec{S}^*$ can be feasibly scheduled with completion time $r_i + s_i$, and that $r_i + s_i \leq d_{j_i}$. We know this holds since $\mathcal{O}$ is a feasible schedule so j_i is scheduled with completion time at most its deadline, and $r_i + s_i$ is defined by the completion time of j_i when we $\vec{S}^*$ is defined in subsection 4.1.1. We lastly check that straddle jobs will not be processing at the same time, which obviously holds for $\vec{S}^*$ since it is defined from a feasible schedule.

To see that $\vec{a}^*$ is chosen by our algorithm, we need to see that it passes the checks given that we have chosen $\vec{S}^*$. The first check is simply that $a^*_{i,t} = 0$ if the release interval i is earlier than the release time or at least the deadline interval of type t. This check clearly passes since each $a_{i,t}$ is defined whether jobs of type t are scheduled in the release interval i and release interval i is no the deadline interval. We next check if the sum of allotments for

each release interval i are at most the size of the window they are assigned to. This clearly holds since we define the size of each $a_{i,t}$ as the processing time spent on jobs of type t in window i, which must all be entirely between the jobs straddling r_i and r_{i+1}.

□

Lemma 4.1.4. *The runtime of the algorithm is bounded by a polynomial in both n and $p_{\max}$.*

Proof. For fixed choice of straddle jobs $\vec{S}$, we first bound the size of the table A_i. The number of job types is bounded by $O(R^2)$, so the vector $\vec{a}_i$ is bounded by $O(R^2)$ since it is indexed by the number of job types. Therefore, the size of $\vec{a}$ is bounded by $O(R^3)$ since it is $O(R)$ many vectors $\vec{a}_i$. So the index $\vec{a}$ in the table A_i, is bounded by $O(R^3)$. For a fixed choice of straddle job (j_i, s_i), the number of indices l in $A_i[\vec{b}, j_q, l]$ is bounded by $O(np_{\max})$ since we force l to be no more than $np_{\max}$ larger than $r_i + s_i$. Lastly, the index j_q is bounded by a polynomial in n since there are at most n jobs. So the size of the table is bounded by a polynomial in n and $p_{\max}$.

We next need to see that the number of straddle jobs and allotments we iterate over is bounded by a polynomial in n and $p_{\max}$. Each allotment $a_{i,t}$ has $np_{\max} + 1$ possible sizes to choose from, and we know there are $O(R^3)$ many indices of $\vec{a}$, therefore, there are $O((np_{max})^{R^3})$ many choices for $\vec{a}$. The vector $\vec{S}$ has at most R many entries, where each entry picks from $O(n)$ many jobs, and each entry has $O(p_{\max})$ many choices for completion time. Therefore, there are $O((np_{\max})^R)$ many choices for $\vec{S}$.

Next, we see that we can compute the value of a table entry in polynomial time since we attempt to schedule j_q in at most $O(R^2)$ places. Finally, we check that the schedule is feasible in polynomial time, since we check that all $O(n)$ scheduled jobs are scheduled feasibly. □

Theorem 4.1.5. *This algorithm gives an exact solution for* Throughput Maximization *with a constant number of release times.*

4.2 Constant Span

The last and most general case of Throughput Maximization we consider is for any job j, the number of release times between the release time and deadline of that job is bounded by a constant value. We call the number of such release times the *span* of a job, and let $span_j$ be the variable we will use to describe this in the Graham notation. We prove the following Theorem

Theorem 4.2.1. *There is an algorithm that finds an solution to $1|span_j \in O(1)|\sum_j U_j$ and runs in time polynomial in n and $p_{\max}$.*

This generalizes the results from Section 4.1 since we do not restrict the number of release times, just the span of jobs. The definitions of job types, windows, straddle jobs, and allotments from Section 4.1 are kept the same. To build our algorithm, given a fixed choice of straddle jobs and allotments, we will apply insight from Section 4.1 that we can independently find a schedule for jobs that share a deadline window. Letting α be the maximum span of release times, our algorithm will use this insight by guessing the optimal straddle jobs and allotments for the fixed range of straddle points $\{r_{i-\alpha}, \dots, r_i, r_{i+1}\}$, and apply the algorithm from Section 4.1 to find A_i.

It is clear that the algorithm of Section 4.1 finds an exact solution to this case. However, this would run in time $\Omega(n^R)$, since R is not assumed to be constant for this case this algorithm is not even pseudopolynomial time. This is because the number of straddle jobs and allotments that algorithm iterates over is at least $\Omega(n^R)$, however the time to compute A_i is unaffected since we assume a constant sized span, therefore, there are only a constant number of job types with deadline interval i.

Therefore, our algorithm is a sweep over windows from left to right, at each step keeping a memory of the last α straddle jobs and allotments. The stored straddle jobs and allotments can be used to compute the value for A_i, since the jobs with deadline in this window would have been released in the last α windows, and so have well defined allotments to schedule in. To push the window forward, we simply guess an optimal straddle job and allotments for the next release time and add this guess to our memorized set of straddle jobs and allotments, forgetting the straddle job and allotments for the earliest release window if we have memorized more than α windows.

4.2.1 Canonical Schedules

Given optimal schedule $\mathcal{O}$, we can sort it in the same way a canonical optimal schedule from subsection 4.1.1, in that the schedule is broken into straddle jobs and allotments, and the jobs in their deadline interval are scheduled in increasing order of deadline right-shifted to the start time of the allotments. As in 4.1.1 we can denote the straddle jobs of $\mathcal{O}$ with $\vec{S}^*$ and the allotments of job types not in their deadline window as $\vec{a}^*$, where jobs that are scheduled in allotments are grouped according to type.

As in subsection 4.1.1 we know we can compute A_i independently of all jobs with deadline window not equal to $[r_i, r_{i+1}]$ given a fixed choice of straddle jobs and allotments since we only need the straddle jobs and allotments for straddle points $\{r_{i-\alpha}, \dots, r_{i+1}\}$ to find a schedule. This gives us a more useful observation

Observation 4.2.1. *Job types with deadline window $[r_i, r_{i+1}]$ can be scheduled independently of all other job types given a choice of straddle jobs and allotments for straddle points $\{r_{i-\alpha}, \dots, r_i, r_{i+1}\}$.*

This observation is key for our dynamic program, where we guess the optimal straddle jobs and allotments for just these α windows to compute A_i using the algorithm of Section 4.1.

4.2.2 Algorithm

The algorithm here is a dynamic program that is a "sliding window" across release times where we guess the optimal straddle jobs and allotments, and then run the dynamic program from Section 4.1 to find an optimal schedule for these choices of straddle jobs and allotments. Specifically, for each release time r_i in increasing order, we guess the optimal choice of jobs straddling release times $\{r_{i-\alpha}, \ldots, r_i, r_{i+1}\}$, which we denote $\vec{S}^i \in (\mathcal{J} + \bot)^{\alpha+1} \times [0, p_{\max}]$, and guess the optimal allotments for the resulting windows, namely $\vec{a}^i$. We check that these choices correspond to canonical schedules in the same way as in Section 4.1, ensuring that straddle jobs are scheduled feasibly and that there is enough interval between straddle jobs for the size of allotments.

For each choice of $\vec{S}^i$ and $\vec{a}^i$, compute the table A_i using the algorithm from Section 4.1. We then call on the previously computed schedule for jobs with earlier deadline interval by guessing an optimal choice for $\vec{S}^{i-1}$ and $\vec{a}^{i-1}$, where the last $\alpha - 1$ entries of $\vec{S}^{i-1}$ equal the first $\alpha - 1$ entries of $\vec{S}^i$, that is, they are equal at every release time except $r_{i-\alpha-1}$ and r_{i+1}. Similarly, we assume that the last $\alpha - 1$ entries of $\vec{a}^{i-1}$ are equal to the first $\alpha - 1$ entries of $\vec{a}^i$, that is, except for the windows in $[r_{i-\alpha-1}, r_{i-\alpha}]$ and $[r_i, r_{i+1}]$ the allotments are equal for each window.

Our algorithm computes the table $B[\vec{a}^i, \vec{S}^i]$, which stores the optimal solution up to release interval i, given stored straddle jobs and allotments $\vec{S}^i$ and $\vec{a}^i$.

For a release interval i, $\mathcal{J}_i$ denotes the jobs with deadline in interval i, and $\mathcal{J}_i(S)$ denotes the jobs from $\mathcal{J}_i$ not scheduled after scheduling straddling jobs $\vec{S}^i$. We sort the jobs in $\mathcal{J}_i(S)$ by deadline and enumerate them as $\mathcal{J}_i(S) = \{j^i_1, \ldots, j^i_{|\mathcal{J}_i(S)|}\}$. As in Section 4.1 we let $L = \min\{L_i, r_i + s_i + np_{\max}\}$ denote the smaller of the start time of the allotments in window i, or $np_{\max}$ past the straddle job, so we can bound runtime. We then use this to compute the table $A_i\left[\vec{a}, j^i_{|\mathcal{J}_i(S)|}, L\right]$. Thus the recursive formula for this algorithm will be

$$B[\vec{a}^i, \vec{S}^i] = \max_{\vec{S}^{i-1}, \vec{a}^{i-1}} \left[B[\vec{a}^{i-1}, \vec{S}^{i-1}]\right] + A_i\left[\vec{a}^i, j^i_{|\mathcal{J}_i(S)|}, L\right]$$

4.2.3 Algorithm Analysis

Claim 4.2.2. *The algorithm finds an optimal solution.*

Proof. This claim is shown by induction on i. For $i = 1$, we only need to schedule A_1 which we can do optimally with the schedule from Section 4.1, so $B[\vec{a}^1, \vec{S}^1]$ is optimal.

If we have computed B_{i-1}, then we just to need to see that we find the optimal A_i and call on the optimal B_{i-1} to get the solution for B_i. This holds since we iterate across all choices of $\vec{a}^i$ and $\vec{S}^i$, and so we find the optimal A_i. We can then iterate across all choices of $\vec{a}^{i-1}$ and $\vec{S}^{i-1}$ that are equal to $\vec{a}^i$ and $\vec{S}^i$ for $\alpha - 1$ straddle points. With these we can recursively call on values for B_{i-1}, which we know are optimal by assumption. □

Claim 4.2.3. *The runtime of the algorithm is polynomial in n and $p_{\max}$.*

Proof. For fixed release interval i, we know by the proof of Lemma 4.1.4 that the number of choices for both $\vec{S}^i$ and $\vec{a}^i$ is bounded by a polynomial in n and $p_{\max}$ since they are defined over a constant number of release times. This also shows that there are a polynomial number of choices for $\vec{S}^{i-1}$ and $\vec{a}^{i-1}$. Lastly, we know that we can compute the table A_i in polynomial time since we are only computing for a constant number of job types. □

Theorem 4.2.4. *This algorithm gives an exact solution for* Throughput Maximization *with job release times and deadlines spanning a constant number of release times.*

4.2.4 Extending to a Constant Number of Machines

Here we describe how to extend the results of this chapter to a constant number of machines to get a pseudo-polynomial time algorithm for both $Pm|\#r_j \in O(1)|\sum_j U_j$ and $Pm|\#size_j \in O(1)|\sum_j U_j$. The extension is very similar to the extension from Chapter 3 in that we will give a natural definition for straddle jobs and allotments on multiple parallel machines, and provide a natural extension to the algorithm already provided. Straddle jobs are tuples $(i, j, s_k) \in \{1, \ldots, m\} \times \{\mathcal{J} + \perp\} \times [0, p_{\max} - 1]$, where i denotes the machine the job j is scheduled on, and j has completion time $r_k + s_k$, where the release time r_k is inferred by the index of the straddle job as before. Given the straddle jobs, for a fixed machine we can define allotments in the same way as before, so we have an allotment vector $\vec{a}^i$ for each machine and an allotment vector $\vec{a}$ for the whole schedule.

We can define canonical schedules in basically the same way as for single machines in that we consider an optimal schedule $\mathcal{O}$ and for each machine we do the rescheduling specified in Subsection 4.1.1 so that the jobs on each machine are written as straddle jobs, allotments, and jobs in their deadline window. Our algorithm will be almost identical to the single machine version, given guesses for allotments and straddle jobs, for each deadline window sort the jobs by deadline and schedule them in the same order and manner as in the single machine case, except that when scheduling a job we also guess which of the m machines it will be scheduled on. When making guesses for optimal straddle jobs and allotments, we check that the choices for each machine correspond to a canonical schedule for that machine.

When scheduling a job we make a factor of $m \in O(1)$ more guesses for where a job should be scheduled since we just over every machine. Since we find allotments across m

machines the size of $\vec{a}$ is bounded by $O(mR^3) = O(1)$, so the number of choices for $\vec{a}$ is still bounded by $(np_{\max})^{O(1)}$. The vector of straddle jobs $\vec{S}$ has no more than mR entries, and each entry picks from $O(n)$ jobs with $O(p_{\max})$ choices for completion time, therefore there is $O((np_{\max})^{mR})$ many possible choices for $\vec{S}$. Lastly, scheduling a job still takes $O(1)$ time since we just try to place it in one of $O(mR^2) = O(1)$ many places.

Since we have a pseudo-polynomial time algorithm for $Pm|\#r_j \in O(1)|\sum_j U_j$ we can use this as a subroutine to find a pseudo-polynomial time algorithm for $Pm|span_j \in O(1)|\sum_j U_j$. As in the single machine case if we have a choice of straddle jobs and allotments for straddle points $\{r_{i-\alpha}, \ldots, r_{i+1}\}$ we can find an optimal schedule for the jobs with deadline interval i. So our algorithm will be much the same as in the single machine case, we perform a left to right sweep keeping a memory of the last α straddle points, use the algorithm for $Pm|\#r_j \in O(1)|\sum_j U_j$ to schedule the jobs in the latest deadline interval, then iterate across straddle jobs and allotments for the first release interval not memorized.

Chapter 5

Conclusion

In this thesis we presented several algorithms for various cases of THROUGHPUT MAXIMIZATION .

We presented an exact polynomial time algorithm for $1|\#r_j \in O(1), \#p_j \in O(1)|\sum_j U_j$ in Chapter 2, which made extensive use of the Jackson rule to find a schedule between release times. This algorithm was an attempt at closing a generalization of a problem identified as open by Sgall [20].

In Chapter 3 we found both an exact pseudo-polynomial time algorithm and a PTAS for $Pm|\#r_j \in O(1), \#d_j \in O(1)|\sum_j U_j$ by developing the idea of jobs either straddling release times and deadlines or scheduled between these points. We also showed this problem is NP-Complete by a reduction to the PARTITION problem. The algorithms found in this Chapter rely on a simple algorithm for the MULTIPLE KNAPSACK problem that is written out in Appendix A.

This straddle job and allotment paradigm was of use in finding an exact pseudo-polynomial time algorithm for both $Pm|\#r_j \in O(1)|\sum_j U_j$ and $Pm|\#size_j \in O(1)|\sum_j U_j$, which is the most general result of this work.

Some future direction of research in this area involves

- An obvious potential direction of research is in extending $1|\#r_j \in O(1), \#p_j \in O(1)|\sum_j U_j$ to either more machines, or relaxing the requirements on the number of release times. The requirement that there are a constant number of unique release times is useful in the algorithm we presented to allow us to efficiently find the jobs that are available at a release interval.

- The case of $1|r_j, p_j \leq O(1)|\sum_j U_j$ was in fact pursued during the course of this research. The intended approach was to break the maximum timeline into a constant number of segments and rounding the release time and deadline span of jobs with large spans to exactly correspond to these segment and then try to guess how many of these jobs are scheduled in each segment, and then recurse to each segment. This

direction was complicated by the fact that if a jobs release time and deadline spanned only a few of these segments then this rounding does not appear to work out.

- Another obvious research direction is pursuing a PTAS for both $Pm|\#r_j \in O(1)|\sum_j U_j$ and $Pm|\#size_j \in O(1)|\sum_j U_j$, possibly by extending the pseudo-polynomial time algorithm given. This direction was considered for a single machine, even for $O(\log n/\log\log n)$, but a major sticking point was in scheduling jobs in their deadline interval which has size polynomial in job sizes; since jobs cannot be rescheduled in arbitrary order like in the proof for 3.3.1, we cannot easily round the size of guesses for deadline interval.

References

[1] P. Baptiste, “Polynomial time algorithms for minimizing the weighted number of late jobs on a single machine with equal processing times,” *Journal of Scheduling*, vol. 2, no. 6, pp. 245–252, 1999. 10

[2] P. Baptiste, P. Brucker, S. Knust, and V. G. Timkovsky, “Ten notes on equal-processing-time scheduling,” *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, vol. 2, no. 2, pp. 111–127, 2004. 10

[3] A. Bar-Noy, S. Guha, J. Naor, and B. Schieber, “Approximating the throughput of multiple machines in real-time scheduling,” *SIAM Journal on Computing*, vol. 31, no. 2, pp. 331–352, 2001. 9

[4] P. Brucker, *Scheduling algorithms*. Springer, 2007, vol. 3. 1, 4, 10

[5] P. Brucker and S. A. Kravchenko, “Scheduling equal processing time jobs to minimize the weighted number of late jobs,” *Journal of Mathematical Modelling and Algorithms*, vol. 5, no. 2, pp. 143–165, 2006. 10

[6] C. Chekuri and S. Khanna, “A polynomial time approximation scheme for the multiple knapsack problem,” *SIAM Journal on Computing*, vol. 35, no. 3, pp. 713–728, 2005. DOI: 10.1137/S0097539700382820. [Online]. Available: https://doi.org/10.1137/S0097539700382820. 7, 23–26

[7] J. Chuzhoy, R. Ostrovsky, and Y. Rabani, “Approximation algorithms for the job interval selection problem and related scheduling problems,” *Mathematics of Operations Research*, vol. 31, no. 4, pp. 730–738, 2006. 9

[8] M. C. Dourado, R. de Freitas Rodrigues, and J. L. Szwarcfiter, “Scheduling unit time jobs with integer release dates to minimize the weighted number of tardy jobs,” *Annals of Operations Research*, vol. 169, no. 1, pp. 81–91, 2009. 10

[9] M. R. Garey and D. S. Johnson, “Two-processor scheduling with start-times and deadlines,” *SIAM Journal on Computing*, vol. 6, no. 3, pp. 416–426, 1977. 9

[10] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan, “Optimization and approximation in deterministic sequencing and scheduling: A survey,” in *Annals of discrete mathematics*, vol. 5, Elsevier, 1979, pp. 287–326. 4

[11] M. Held and R. M. Karp, “A dynamic programming approach to sequencing problems,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 10, no. 1, pp. 196–210, 1962. 9

[12] S. Im, S. Li, and B. Moseley, “Breaking 1-1/e barrier for non-preemptive throughput maximization,” in *International Conference on Integer Programming and Combinatorial Optimization*, Springer, 2017, pp. 292–304. 9

[13] J. Jackson, *Scheduling a production line to minimize maximum tardiness*, ser. Research report. Office of Technical Services, 1955. [Online]. Available: https://books.google.ca/books?id=4jnPJgAACAAJ. ii, 6, 9

[14] D. Karger, C. Stein, and J. Wein, “Scheduling algorithms,” in *Algorithms and theory of computation handbook*, Chapman & Hall/CRC, 2010, pp. 20–20. 9

[15] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*, Springer, 1972, pp. 85–103. 3, 9

[16] E. L. Lawler, "Sequencing to minimize the weighted number of late jobs," *RAIRO Rech. Oper.*, vol. 10, pp. 27–33, 1976. 9

[17] E. L. Lawler and J. M. Moore, "A functional equation and its application to resource allocation and sequencing problems," *Management Science*, vol. 16, no. 1, pp. 77–84, 1969. 9

[18] J. K. Lenstra, A. R. Kan, and P. Brucker, "Complexity of machine scheduling problems," in *Annals of discrete mathematics*, vol. 1, Elsevier, 1977, pp. 343–362. 9

[19] J. M. Moore, "An n job, one machine sequencing algorithm for minimizing the number of late jobs," *Management Science*, vol. 15, no. 1, pp. 102–109, 1968. 6, 9, 12, 16, 17

[20] J. Sgall, "Open problems in throughput scheduling," in *European Symposium on Algorithms*, Springer, 2012, pp. 2–11. ii, 7, 38

[21] F. C. Spieksma, "On the approximability of an interval scheduling problem," *Journal of Scheduling*, vol. 2, no. 5, pp. 215–227, 1999. 9

[22] V. V. Vazirani, *Approximation algorithms*. Springer Science & Business Media, 2013. 1

[23] D. Williamson and D. Shmoys, *The Design of Approximation Algorithms*. Cambridge University Press, 2011, ISBN: 9781139498173. [Online]. Available: `https://books.google.ca/books?id=Cc%5C_Fdqf3bBgC`. 1

Appendix A

Multiple Knapsack Problem for Uniform Profits

Here we give a pseudo polynomial time algorithm for MULTIPLE KNAPSACK Problem with Uniform Profits.

Suppose there n items $j_1, \ldots, j_n$ of uniform profit, $p_i = 1$ for all j_i, and item j_i has weight w_i. Further, say there are $m \in O(1)$ knapsacks with sizes $k_1, \ldots, k_m \in [0, nw_{\max}]$. We let the table $A[k_1, \ldots, k_m, j_n]$ be the maximum way to place items $j = 1, \ldots, n$ in all the knapsacks.

The algorithm to pack the items is a straightforward dynamic program, where we take the item, and consider packing it in any of the knapsacks or not at all. That is, if we have computed the table for all placements of items $\{j_1, \ldots, j_{q-1}\}$ and for all knapsack capacities $k''_i \leq k'_i$, then we will compute the table $A[k'_1, \ldots, k'_m, j_q]$. If j_q is not part of the optimal solution, then the optimal solution is composed of the jobs $\{j_1, \ldots, j_{q-1}\}$, so we have $A[k'_1, \ldots, k'_m, j_q] = A[k'_1, \ldots, k'_m, j_{q-1}]$. If j_q is part of the optimal solution, then it is packed in some knapsack k'_i, and we reduce the capacity of knapsack i by w_{j_q} so $A[k'_1, \ldots, k'_m, j_q] = A[k'_1, \ldots, k'_i - w_{j_q}, k'_m, j_{q-1}]$. If $q = 1$, then the optimal packing is placing j_1 in any knapsack where it would fit. This gives the following lemma

Lemma. *This algorithm finds an optimal solution.*

Lemma. *This algorithm runs in time polynomial in n and $w_{\max}$.*

Proof. The largest size of a knapsack is $nw_{\max}$, so there are $(nw_{\max})^m$ many indices for knapsacks in the table. There are exactly n items to consider, therefore, the table has $n(nw_{\max})^m = (nw_{\max})^{0(1)}$ many indices. Placing an item takes $O(1)$ time since we only consider whether it can fit in one of the $O(1)$ knapsacks. □