Fully Polynomial Time Approximation Schemes for Sequential Decision Problems

by

Mohamed Mostagir

Submitted to the Sloan School of Management
in partial fulfillment of the requirements for the degree of

Master of Science in Operations Research

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

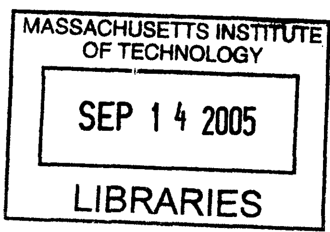
September 2005

© Massachusetts Institute of Technology 2005. All rights reserved.

Author ....
Sloan School of Management
Aug 12, 2005

Certified by ....
James B. Orlin
E. Pennel Brooks Professor of Operations Research
Thesis Supervisor

Accepted by ....
John N. Tsitsiklis
Professor of Electrical Engineering and Computer Science
Co-Director, Operations Research Center



Fully Polynomial Time Approximation Schemes for Sequential Decision Problems

by

Mohamed Mostagir

Submitted to the Sloan School of Management
on Aug 12, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science in Operations Research

Abstract

This thesis is divided into two parts sharing the common theme of fully polynomial time approximation schemes. In the first part, we introduce a generic approach for devising fully polynomial time approximation schemes for a large class of problems that we call list scheduling problems. Our approach is simple and unifying, and many previous results in the literature follow as direct corollaries of our main theorem. In the second part, we tackle a more difficult problem; the stochastic lot sizing problem, and give the first fully polynomial time approximation scheme for it. Our approach is based on simple techniques that could arguably have wider applications outside of just designing fully polynomial time approximation schemes.

Thesis Supervisor: James B. Orlin
Title: E. Pennel Brooks Professor of Operations Research

Acknowledgments

I would like to thank my advisor, Jim Orlin, for his unique insights into the problems we worked on, and for teaching me to make my proofs simpler and more concrete. There is an Einstein quote which I believe summarizes Jim's approach to research, "Everything should be made as simple as possible, but not simpler."

It was wonderful to be able to interact with David Simchi-Levi, a great teacher and a great person. David has this very reassuring quality that makes you feel that he is strongly rooting for you to succeed. Eventhough our meetings were few and far between, he would remember exactly everything (down to the technical details) we discussed in our last meeting, even if it was more than a month ago!

I have too many friends to thank. I would like to thank Amr for his friendship during the year we overlapped at the ORC, Nelson for all the couch conversations, and David for the music we played. I would like to thank Hamed for too many funny moments in Tang and after, and for his help during times of difficulty. I would like to thank his wife, Rosa, as well. I would like to thank Theo for innumerable conversations about just everything, agreeing with me on every topic possible, and for helping this thesis make the deadline (via moral support from Mexico). I would like to thank Marwa for teaching me to get my priorities in order, albeit in a very hard way.

Finally, I would like to thank my parents and my sister Mai for their unconditional love and for always being with me in spirit.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

1 Fully Polynomial Time Approximation Schemes for Sequential Optimization Problems 9

1.1 Introduction 9
1.2 Branch and Dominate for Combinatorial Optimization Problems . . . 12
1.3 Branch and δ-Dominate 18
1.4 Applications Of Main Theorem 24
1.4.1 Machine Scheduling Problems 24
1.4.2 Scheduling With Dependent Processing Times 26
1.4.3 Single Machine Problems 28
1.4.4 Maximization Problems and Polynomial Objective Functions . 29
1.5 Comparisons With Woeginger 31
1.6 Multi - Criteria problems 34
1.6.1 Minimizing Makespan and Sum of Cubed Completion Times on Two Machines 36
1.6.2 Problems with 'Crashing' 36
1.7 Summary Of Results for Scheduling Problems 40
1.7.1 Problems With an FPTAS 41
1.8 Conclusions 43

2 Stochastic Lot Sizing 45

2.1 Introduction 45
2.2 Models And Assumptions 47
2.3 Pseudo-Polynomial Time Algorithms 49

2.3.1 Linear plus fixed costs 51
2.4 Approximation Schemes 53
2.4.1 Approximating $Y_j(z)$ 57
2.4.2 Approximating $g_j(I)$ 59
2.4.3 Approximation guarantee and running time analysis 61
2.5 Conclusions and Future Work 63

Chapter 1

Fully Polynomial Time Approximation Schemes for Sequential Optimization Problems

1.1 Introduction

In this chapter, we combine several approaches for developing fully polynomial time approximation schemes into a generic method. Our objective is to develop a simple framework that unifies a large number of results in the area of fully polynomial time approximation schemes. We then apply our framework to give single and multiple criteria FPTASes for a wide range of machine scheduling problems that we refer to as "list scheduling problems."

Assume that we have an NP-hard problem, [1] and an approximation algorithm that always returns a near-optimal solution whose cost is at most a factor of α away from the optimal cost, where $\alpha > 1$ is some real number: In minimization problems the near-optimal cost is at most a multiplicative factor of α above the optimum, and in maximization problems it is at most a factor of α below the optimum. Such an approximation algorithm is called an *alpha*-approximation algorithm. A family of $(1 + \epsilon)$-approximation algorithms over all $\epsilon > 0$ is called a fully polynomial time

approximation scheme or FPTAS, for short, if the time complexity is bounded by a polynomial in the size of the input and in $\frac{1}{\epsilon}$.

The first work done on fully polynomial time approximation schemes could be tracked back to the mid-70s, starting with the classic work of Horowitz, Ibarra, Kim, and Sahni [2],[3],[4] on scheduling and the knapsack problems. Those first FPTASes in [2], [3], and [4] were based on dynamic programming formulations which can always find an exact optimal solution, but in pseudo-polynomial time. Our approach, branch and dominance, is similar in spirit to dynamic programming, especially in the way we define and use states. The primary goal of our work was to simplify the research done by Woeginger in [5]. Woeginger's work aims to construct a generic framework around dynamic programming formulations and to identify a certain class of those formulations that he called 'DP (short for dynamic programming)- Benevolent'. A problem is DP - Benevolent if it has a dynamic programming formulation that satisfies certain arithmetical and technical conditions. Woeginger then showed that it is easy to find FPTASes for this class of problems. Even though our initial goal was to design an easier approach to obtain FPTASes, our work expanded to obtain FPTASes for multi-criteria problems as well, where one is interested in computing a tradeoff curve, known as the Pareto curve [6], between the various criteria being optimizes. We show how to extend our approach to give ϵ-approximate Pareto curves. We also show how to obtain FPTASes for a certain class of problems where the cardinality of the decision set is exponential. These extensions were not considered by Woeginger.

In addition to giving a simpler approach to constructing FPTASes than that in [5], our work was also motivated by its application to a wide class of machine scheduling problems. Many scheduling problems fall into the framework we construct in this chapter and hence we can derive FPTASes for them (For example, [7], [2], [3], [8].) We consider a class of problems that we call '*list scheduling*' problems, where the input to the problem consists of a list of the jobs to be scheduled, and where the jobs should be processed in the order in which they appear on the list. Many of the results in the scheduling literature show that one can get an optimal solution to a scheduling problem if one first arranges the jobs to be scheduled in a certain order

before processing them. Hence, an optimal solution to the list scheduling problem automatically gives a solution to the general problem. Additional applications can be found in very large scale neighborhood search [9], where a local search algorithm starts with a feasible solution to an optimization problem and iteratively tries to improve that solution by searching the 'neighborhood' of that solution. In very large scale neighborhood search, the size of the neighborhood could be very large relative to the input data, and it becomes impractical to search the whole neighborhood. Thus at every solution, one should try and search the neighborhood efficiently by considering only a subset of all the potential solutions in the neighborhood.

We consider combinatorial optimization problems $min(c(x) : x \in P)$, where x is a decision vector with n components. Our approach is based on "branch and dominance." We start with an enumeration tree, where each node of the tree corresponds to a partial solution $y = y_1, ..., y_k$ for some value of k. Each partial solution y is associated with a state vector (as in dynamic programming), called $State(y)$. We also assume that there is a dominance relation $\preceq$ on states such that $State(y) \preceq State(w)$ implies the following: if w can be extended to a feasible (complete) solution w', then y can be extended to a feasible complete solution y' with $c(y') \leq c(w')$. Branch and dominate is the algorithm in which nodes of the enumeration tree are enumerated in a breadth first search manner, and nodes that are dominated are eliminated from the tree. We give conditions under which Branch and Dominate leads to a pseudo-polynomial time algorithm.

For a given value of ε we show how to approximate $State(y)$ using geometric rounding, resulting in δ-$State(y)$, where δ is a parameter that depends on ϵ and on n. We then consider Branch and δ-dominate, which is the same as Branch and Dominate, except that we eliminate partial solutions y' from the tree whenever there exists another partial solution y such that δ-$State(y) \preceq \delta$-$State(y')$. We give sufficient conditions under which Branch and δ-dominate is an FPTAS. Initially, we assume that there is a single objective function, and we assume that there is an upper bound on y_k that is polynomially bounded in n. Subsequently, we show how to relax these assumptions. We develop FPTASes for combinatorial optimization problems with a

fixed number of criteria, and we permit decision variables to take on an exponentially large number of different values.

The remainder of this chapter is organized as follows. In Section 1.2 we introduce Branch and Dominate and give conditions under which it gives a pseudo-polynomial time algorithm. Section 1.3 gives extra conditions under which Branch and Dominate (now called Branch and δ-Dominate) gives an FPTAS. Section 1.4 illustrates various examples on how to apply the main result. Section 1.5 reviews the approach taken by Woeginger and compares and contrasts our approach with of his. Section 1.6 extends our work to problems where we are interested in the tradeoffs when optimizing more than one criteria and where the possible decisions that we can take at any stage of the problem is exponential. Section 1.7 give a summary of various scheduling problems that fall into the branch and dominance framework and Section 1.8 concludes the chapter.

1.2 Branch and Dominate for Combinatorial Optimization Problems

A *combinatorial optimization problem* Π is a collection of instances $(\mathcal{X}, c)$, where $\mathcal{X}$ is a collection of feasible solutions and c is an objective function which, for every solution $x \in \mathcal{X}$, assigns a non-negative cost $c(x)$. The goal is to find a solution x^* with $c(x^*) \leq c(x), \forall x \in \mathcal{X}$. A *0-1 combinatorial optimization problem* is a combinatorial optimization problem in which every feasible solution $x \in \mathcal{X}$ is a vector of 0's and 1's. In the following, we assume that the number of components of x is n. A *bounded combinatorial optimization problem* is a combinatorial optimization problem with an associated vector u of upper bounds. If $x \in \mathcal{X}$, then $0 \leq x \leq u$, and x is integer valued. In this chapter, we consider both bounded combinatorial optimization problems and their special case, $0-1$ combinatorial optimization problems.

A *partial solution* is a vector $x' = x'_1, ..., x'_k$ that specifies the first k components of a solution x. Our algorithm will enumerate solutions by determining partial

solutions one component at a time. A partial solution with k components will be called a solution at *stage* k. Suppose that x' is a partial solution at stage k, and that $y' = y'_{k+1}, ..., y'_r$ is a specification of values for components $k+1$ to r. We let $x' \blacklozenge y' = x'_1, ..., x'_k, y'_{k+1}, ..., y'_r$ be the concatenation of x' and y'. If $r = n$, and if $x' \blacklozenge y' \in \mathcal{X}$, then we refer to y' as a *feasible completion* of x'.

We assume that there is a function h_k which can be used to test the feasibility of a partial solution x' at stage k. If $h_k(x') = False$, then no completion of x' is in $\mathcal{X}$. If $h_k(x') = True$ for $1 \leq k \leq n-1$, there may or may not be a feasible completion of x'. In addition, $h_n(x') = True$ if and only if $x' \in \mathcal{X}$. In other words, h_k gives sufficient but not necessary conditions for a partial solution being infeasible, but it gives necessary and sufficient conditions for a $0-1$ vector of n components to be feasible.

An *enumeration tree* for $(\mathcal{X}, c)$ consists of $2^{n+1}-1$ nodes. The *root node* corresponds to the null solution, and the root is said to be at *stage* 0. Each node at *stage* k will correspond to a *partial solution* $x' = x'_1, ..., x'_k$, where $0 \leq x'_i \leq u_i$ for $i = 1$ to k and x'_i is integral. If $k < n$, x' will have $u_i + 1$ children: $x'_1, ..., x'_k, x'_{k+1}$ with x'_{k+1} taking on integer values from 0 to u_i. We refer to node x' as the *parent node* of each of these $u_i + 1$ children. The nodes at stage n have no children, and are the *leaves* of the enumeration tree. If $x' = x'_1, ..., x'_k$ is a partial solution at stage k, and if $y' = y'_{k+1}, ..., y'_n$ is a partial assignment for variables $k+1$ to n, we refer to y' as a *completion* of x'. We let $x' \blacklozenge y' = x'_1, ..., x'_k, y'_{k+1}, ..., y'_n$ denote the *concatenation* of x' and y'.

In the enumeration tree, we permit the enumeration of infeasible solutions. In the algorithm developed later in this section, we will eliminate any infeasible nodes.

Central to our approach is the concept of a *state*, which is used in the standard way as in dynamic programming. Associated with each node x' of the enumeration tree is a state vector $State(x')$. We assume that state vectors satisfy the following conditions:

Condition 1. *(State Vector Conditions.)*

1. For every partial solution x', each component of $State(x')$ is a non-negative integral vector.

2. If x' and x'' are two partial solutions at stage k for $k = 1$ to n, and if $State(x') = State(x'')$, then $h_k(x') = h_k(x'')$;

3. If x' and x'' are two partial solutions at stage $k-1$ for $1 \leq k \leq n$ and if $State(x') = State(x'')$, then for $0 \leq y_k \leq u_k$ and y_k integral, $State(x' \blacklozenge y_k) = State(x'' \blacklozenge y_k)$.

4. The first component of the state vector represents the objective function. So, if x' is a feasible solutions at stage n, and if $S = State(x')$, then $S_1 = c(x')$.

Condition 2. *(Computability Conditions.)*

1. *The state vector $State(x')$ can be computed in polynomial time for every partial solution x'.*

2. *The function $h_k(x')$ can be computed in polynomial time for each $k = 1$ to n,and for every partial solution x' at stage k.*

3. *There is a guaranteed upper bound on the value that any component can take. This upper bound is a polynomial in the input size of the problem and the largest integer in the data.*

Thus if two partial solutions have the same state, then one need not enumerate children from both nodes in the enumeration tree. It suffices to enumerate children from one of the nodes only.

Because stage-n solutions with the same state vector have the same cost, we use the following convention. If $S = State(x')$,we let $c(S) = c(x')$. Although this is an abuse of notation, it should be clear from context whether we are evaluating the cost of a state vector at stage n or whether we are evaluate the cost of a feasible solution.

Let the set of all state vectors in stage k be denoted by $\mathbb{S}_k$. Let $\mathcal{F}_k$ be a mapping from the state vectors in $\mathbb{S}_{k-1}$ to the state vectors in $\mathbb{S}_k$ defined as follows. Let x' be

a partial solution at stage $k-1$, and let $y_k \in \{0, 1, ..., u_k\}$. Then $\mathcal{F}_k(State(x'), y_k) = State(x' \blacklozenge y_k)$. From Condition 1 above, $\mathcal{F}_k$ is well defined, and by Condition 2, $\mathcal{F}_k$ can be computed in polynomial time. Let $\mathcal{F}_{ki}$ denote the i-th component of $\mathcal{F}_k$. For example, if $\mathcal{F}_k(S, y_k) = S' = S'_1, ..., S'_\alpha$ then $\mathcal{F}_{ki}(S, y_k) = S'_i$.

In order to develop a concept of "domination", we will make additional assumptions on the state vectors.

Condition 3. *For a given combinatorial optimization problem, all state vectors have the same number α of components.*

If for some i and for every k the domain for the i-th component of the state vector is polynomially bounded in n, then we say that component i of the state vector is *polynomially bounded*, or *PB* for short.

Definition 1. *We say that a (non-polynomially bounded) component i has the* weak lower-is-better property *if the following is true: For all state vectors $S = S_1, ..., S_\alpha$ and $S' = S'_1, ..., S'_\alpha$ at stage k with $S_j = S'_j$ for $j \neq i$, and $0 \leq S_i \leq S'_i$,*

1. if $h_k(S) = false$, then $h_k(S') = false$;
2. if $k = n$, then $c(S) \leq c(S')$.

Definition 2. *We say that a (non-polynomially bounded) component i has the* weak higher-is-better property *if the following is true: For all state vectors $S = S_1, ..., S_\alpha$ and $S' = S'_1, ..., S'_\alpha$ at stage k with $S_j = S'_j$ for $j \neq i$, and $0 \leq S_i \leq S'_i$,*

1. if $h_k(S) = true$, then $h_k(S') = true$;
2. if $k = n$, then $c(S) \geq c(S')$.

Definition 3. *A non polynomially bounded component i is called "monotone" if it has the additional properties: For all state vectors $S = S_1, ..., S_\alpha$ and $S' = S'_1, ..., S'_\alpha$ at stage $k-1$ with $S_j = S'_j$ for $j \neq i$ and for all y_k, it follows that*

1. If $0 \leq S_i \leq S'_i$ and if component i has the weak lower-is-better property, then $\mathcal{F}_{kj}(S, y_k) \leq \mathcal{F}_{kj}(S', y_k)$ for every component j with the weak lower-is-better property;

2. If $0 \leq S_i \leq S'_i$ and if component i has the weak higher-is-better property, then $\mathcal{F}_{kj}(S, y_k) \geq \mathcal{F}_{kj}(S', y_k)$ for every component j with the weak higher-is-better property;

3. $\mathcal{F}_{kj}(S, y_k) = \mathcal{F}_{kj}(S', y_k)$ for each component $j \in PB$;

A component is said to have the *strong lower-is-better* (resp., *strong higher-is-better*) *property* if it has the weak lower-is better (resp., weak higher-is-better) property and if it is also monotone.

Definition 4. *We say that a component i of the state vector influences component j if there are states S and S' at some non-final stage $k-1$ such that S and S' differ only in component i and such that $\mathcal{F}_{kj}(S, y_k) \neq \mathcal{F}_{kj}(S', y_k)$.*

The following algorithm, Branch and Dominate, relies on the following notion of domination. Suppose that the state vectors for a 0-1 optimization problem satisfy Conditions 1, 2, and 3.

Definition 5. *We say that the state vector is strongly monotone if each component either has the strong lower-is-better property, or the strong higher-is-better property, or is polynomially bounded.*

We observe that the polynomially bounded components of the stage vector are permitted to influence all other components. A component with the strong lower-is-better property is permitted to influence only other components with a lower-is-better property. Similarly, a component with the strong higher-is-better property is permitted to influence only other components with a higher-is-better property.

Definition 6. *We say that strongly monotone state vector $S = S_1, ..., S_\alpha$ dominates state vector $S' = S'_1, ..., S'_\alpha$ at stage k if the following is true:*

1. $S_j = S'_j$ for each component j that is polynomially bounded;

2. $S_j \leq S'_j$ for each component j with the strong lower-is-better property; and

3. $S_j \geq S'_j$ for each component j with the strong higher-is-better property.

We next describe Branch and Dominate and give conditions on the state vectors under which Branch and Dominate leads to a pseudo-polynomial time algorithm. Branch and Dominate relies on implicit enumeration, where nodes of the enumeration tree are created in a breadth first search manner, and in which nodes are fathomed (eliminated) from the tree if they are dominated. We will refer to the enumeration tree in which nodes are fathomed as the "partial enumeration tree."

Algorithm. Branch and Dominate

begin

$x_0 = \emptyset$;

initialize the partial enumeration tree T by making its root x_0;

for $k = 1$ to n

begin

for each unfathomed node x' at stage $k-1$, and for each child y of x' for which $h_k(y) = True$, make y the child of x' in T ;

while there are two nodes x' and x'' at stage k such that $State(x')$ dominates $State(x'')$, then fathom node x'';

end

let x^* be a leaf node that minimizes $\{c(x) : x \text{ is a leaf node}\}$;

end

The following Theorem is straightforward, but we include a short proof.

Theorem 1. *Suppose that a combinatorial optimization problem has state vectors satisfying Conditions 1,2, and 3. Suppose further that each component either is*

polynomially bounded, has the strong lower-is-better property, or has the strong higher-is-better property. Then Branch and Dominate determines an optimal solution in pseudo-polynomial time.

Proof. The analysis of the running time is simple. From part 3 of Condition 2, every component at each stage can take a value that is bounded by a polynomial in the input. Let the size of the input be $|I|$, then every stage can have at most $P(|I|)^\alpha$ state vectors, and the number of children is at most $P(|I|)^\alpha \cdot U$. Since α is a constant and we have n stages, the overall running time required to generate the whole enumeration tree is polynomial in $nP(|I|)^\alpha$. This establishes that the running time is pseudo-polynomial. Since only dominated partial solutions are fathomed, the resulting solution to branch and dominate is optimal as well.

□

1.3 Branch and δ-Dominate

An FPTAS is widely considered a very strong approximation result for an NP-hard optimization problem under the assumption that $P \neq NP$. In the previous section, we showed how Branch and Dominate gives a pseudo-polynomial time algorithm for a combinatorial optimization problem under certain conditions. In this section, we modify Branch and Dominate by approximating the state space, using techniques that are very much like those introduced by Ibarra and Kim [3]. We call the resulting algorithm Branch and δ-Dominate. We give additional conditions beyond those mentioned in the previous section under which Branch and δ-Dominate leads to an FPTAS.

For each $\delta \in [0, 1]$, and for each component i, we say that two state vectors S and S' are *component i δ-close* if $(1 - \delta)S'_i \leq S_i \leq (1 + \delta)S'_i$ and if $S_j = S'_j$ for every other component j. If S and S' are component i *δ-close* with respect to all non-polynomially bounded components i, and if $S_j = S'_j$ for every polynomially bounded component j, we say that S and S' are *δ-close.*

Definition 7. *The following three conditions are called the goodness conditions with respect to component i of S.*

a. *Goodness condition 1. If S is a feasible state vector at any stage k, and if S' is obtained from S by replacing component i with a different non-negative integer value, then S' is also feasible state vector;*

b. *Goodness condition 2. If S is a feasible state vector for stage $k-1$, then $S_i \leq F_{ki}(S, y_k)$ for all feasible choices of decisions y_k; that is, the value of the ith component is nondecreasing from stage to stage.*

c. *Goodness condition 3. There exists a parameter $\beta_i > 0$ such that the following is true. For all $\delta \in [0, 1]$, if S and S' are δ-close state vectors with respect to component i at stage $k-1$ for some $k \leq n$, and if x_k is a feasible decision at stage k, then $\mathcal{F}_k(S, x_k)$ and $\mathcal{F}_k(S', x_k)$ are $\beta_i\delta$-close.*

Definition 8. *We say a strongly monotone component i is directly bad if it violates any of the three goodness conditions.*

Definition 9. *We say that a strongly monotone component i is indirectly bad if it is not directly bad, but it influences a bad component.*

Definition 10. *We say that a strongly monotone component i is bad if it is either directly or indirectly bad.*

Note that a component i can be indirectly bad if it influences a component j which in turn influences a directly bad component k. All three components would be called bad.

The bad components are the ones where a small approximation in one stage can lead either to a large relative error in subsequent stages or else can lead to a feasible solution becoming infeasible. To avoid these difficulties, our algorithm will not approximate any of the bad components.

Definition 11. *We say that a strongly monotone component i is good if it is not bad; that is, it satisfies the three goodness conditions and does not influence a bad component.*

Consider the following three examples.

Example 1. Minimize the sum of completion times on two machines subject to exactly K jobs being assigned to machine 1. Here, the state vector for each stage k has 4 components. The first component is the processing time on machine 1 of those jobs from 1 to k that are scheduled on the first machine, and the second component is the processing time on machine 2 of jobs 1 to k scheduled on that machine. The third component is the number of jobs assigned to machine 1. This component is polynomially bounded. The fourth component is the sum of the completion times for the first k jobs. Component 3 is polynomially bounded. Components 1, 2 and 4 are strongly monotone and good, and they have the lower-is-better property.

Example 2. Find the shortest path from node 1 to every other node j with the property that the cost of the path is at most B. The state vector for each stage k has 3 components. Here, the first component in the state vector is the index of the node on the walk of k arcs from node 1. The second component is the length of the walk. The third component is the cost of the walk. The first component is polynomially bounded. The second component is strongly monotone, has the lower-is-better property, and is good. The third component is strongly monotone, has the lower-is-better property, and is bad (because it fails part a of the definition due to the constraint on the total cost).

Example 3. The knapsack problem. The state vector at stage k has two components. The first component is the sum of the values of the items in the knapsack as restricted to items 1 to k. The second component is the weight of the items in the knapsack as restricted to items 1 to k. The first component is strongly monotone, has the higher-is-better property, and is good. The second component is strongly monotone, has the lower-is-better property, and is bad because it violates Condition a of goodness.

Now that the concepts of good and bad monotone states have been formalized, we can proceed to define δ-domination and give the Branch and δ-Dominate algorithm.

Definition 12. *(δ-domination) Suppose that each component is either polynomially*

bounded or strongly monotone. For a given parameter $\delta \in [0,1]$, we say that state vector S δ-dominates state vector S' if the following are true:

1. For all polynomially bounded components i, $S_i = S'_i$;
2. For all bad components i satisfying the lower-is-better property, $S_i \le S'_i$;
3. For all good components i satisfying the lower-is-better property, $S_i \le (1+\delta)S'_i$
4. For all bad components i satisfying the higher-is-better property, $S_i \ge S'_i$;
5. For all good components i satisfying the higher-is-better property, $S_i \ge (1-\delta)S'_i$;

In order to carry out, Branch and δ-Dominate, we replace domination of states by domination of δ-rounded state vector. Let $S = S_1, ..., S_\alpha$ be a state vector. Then, the δ-rounded state vector associated with S is $S^\delta = S_1^\delta, ..., S_\alpha^\delta$, where

$$S_i^\delta = \begin{cases} S_i & \text{if } i \text{ is a polynomially bounded component or a bad component} \\ \lfloor \log_{1+\delta}(S_i+1) \rfloor & \text{if } i \text{ is a good monotone component} \end{cases}$$

The following lemma is an immediate consequence of the definition of the δ-rounded state vector.

Lemma 2. *Suppose that S, T, and W are state vectors at some stage, and let S^δ, T^δ, and W^δ be the corresponding δ-rounded state vectors. If S^δ dominates T^δ, then S δ-dominates T . Moreover, if S^δ dominates T^δ, and if T^δ dominates W^δ, then S^δ dominates W^δ.*

Note that the converse of Lemma 2 is not true. If S δ-dominates T, it is not necessarily the case that S^δ dominates T^δ. Also note that it is possible that S δ-dominates T, and that T δ-dominates W, but that it is not the case that S δ-dominates W. So δ-rounding introduces a transitivity into δ-domination that would not be present without the rounding.

In the following algorithm, we let δ-$State(x)$ denote the δ-rounded state vector of $State(x)$.

Algorithm. Branch and δ-Dominate

begin

$x_0 = \emptyset$;

initialize the partial enumeration tree T by making its root x_0;

for $k = 1$ to n

begin

for each unfathomed node x' at stage $k-1$, and for each child y of x' for which $h_k(y) = True$, make y the child of x' in T ;

while there are two nodes x' and x'' at stage k such that δ-$State(x')$ dominates δ-$State(x'')$, then fathom node x'';

end

let x^* be a leaf node that minimizes $\{c(x) : x \text{ is a leaf node}\}$;

end

Theorem 3. *(Main Theorem) Suppose that* Π *is a* combinatorial optimization problem *in which the number of decisions at each stage is polynomially bounded in the size of the input. Suppose that each solution has an associated state vector satisfying Condition 1. Suppose also that there is at most one bad component, and let* $\beta_{\max}$ $= \max\{\beta_i : i \text{ is a good component}\}$*, where* β_i *is the parameter defined in Condition* 3 *of goodness. Then Branch and* δ*-dominate, with* $\delta = \epsilon/(2\beta_{\max}n)$*, gives an FPTAS for the Combinatorial Optimization Problem.*

Proof. Let ϵ be a positive real number less than 1, and let $\delta = \epsilon/(2\beta_{\max}n)$. We first show that the time taken by Branch and δ-dominate is polynomial in the size P of the input and in $1/\epsilon$, or equivalently, that the time is polynomial in P and in $1/\delta$. We will next show that the number of nodes at Stage k is polynomially bounded in the size of the input and in $1/\delta$. For each polynomially bounded component i of the state vector, let $q_i(P)$ be an upper bound on the number of values that component i can take. If component i is monotone, then let $2^{q_i(P)}$ be an upper bound on the value that component i can take (such a polynomial exists by assumption.)

It follows then that the number of distinct δ-rounded state vectors is $O(\prod_{i=1}^{\alpha} \frac{q_i(P)}{\log(1+\delta)})$, which is polynomial in P and in $1/\delta$. So, at the end of stage k, the number of undominated state vectors is at most the number of boxes, which is polynomial in P and in $1/\delta$. Moreover, since each decision variable can take on only a polynomial number of values, the number of state vectors created at stage k+1 is also polynomially bounded in P and in $1/\delta$. It is clear that all other operations are bounded by a polynomial in P and in $1/\delta$, and so the running time is established.

We next establish that the maximum relative error obtained by the Branch and δ-dominate is $1/\epsilon$. Let $x^* = x_1^*,\ x_2^*,\ \ldots\ ,\ x_n^*$, be an optimal solution for the combinatorial optimization problem. Let y^0, y^1, y^2, ..., y^n, be a sequence of partial solutions chosen as follows: $y^0 = \emptyset$; for each $k = 1$ to n, if $y^{k-1} \blacklozenge x_k^*$ is unfathomed at the end of stage k, then $y^k = y^{k-1} \blacklozenge x_k^*$; otherwise, y^k is an unfathomed node at the end of stage k such that δ-$State(y^k)$ dominates δ-$State(y^{k-1} \blacklozenge x_k^*)$. (The fact that there is some unfathomed partial solution at the end of stage k such that δ-$State(y^k)$ dominates δ-$State(y^{k-1} \blacklozenge x_k^*)$ follows from the fact that $y^{k-1} \blacklozenge x_k^*$ was fathomed, and by the transitivity of the domination operation stated in Lemma 2.

Let $S^k = State(y^k)$ for $k = 0$ to n. Let $T^k = State(y^{k-1} \blacklozenge x_k^*)$. Let $W^k = State(x_1^*, \ldots,\ x_k^*)$. By Lemma 2, S^k δ-dominates T^k for $k = 1$ to n. We now claim inductively on k that the following is true:

1. If component i is a bad mononone component with the higher-is-better property, then $S_i^k \geq W_i^k$;

2. If component i is a bad mononone component with the lower-is-better property, then $S_i^k \leq W_i^k$;

3. If component i is polynomially bounded, then $S_i^k = W_i^k$;

4. If component i is a good mononone component with the higher-is-better property, then $S_i^k \geq (1 - \beta_{\max}\delta)^k W_i^k$;

5. If component i is a good mononone component with the lower-is-better property, then $S_i^k \leq (1 + \beta_{\max}\delta)^k W_i^k$;

It is clearly true for $k = 0$. Assume that the results are true for stage k-1. Then it follows inductively that it is true at stage k from the third condition of goodness. Assume without loss of generality that the objective function is the first component in the state vector and has the lower-is-better property. This implies that at stage n, we have $S_1(y^n) \leq (1 + \beta_{max} \frac{\epsilon}{2\beta_{\max} n})^n S_1(x^*) = (1 + \frac{\epsilon}{2n})^n S_1(x^*)$. We now use the inequality $(1 + \frac{x}{n})^n \leq (1 + 2x)$ for $x \in [0, 1]$, and set $x = \frac{\epsilon}{2}$ to conclude $S_1(y^n) \leq (1 + \epsilon) S_1(x^*)$, which establishes the approximation guarantee and completes the proof.

This result is in the same spirit as that of Woeginger [5], but relies on simple monotonicity rules rather than abstract notions of domination.. We will give a detailed comparison with the approach found in Woeginger in section 1.5.

1.4 Applications Of Main Theorem

In this section, we give some results that follow from Theorem 3. Since our main area of application is scheduling, we give a brief review of the terminology used.

1.4.1 Machine Scheduling Problems

Scheduling a set of jobs on a number of parallel machines to optimize some objective function has been one of the central areas of research in the optimization community for the past thirty years (see for instance [10], [11], and [12]). Throughout that course, results that vary from being very positive (existence of an FPTAS) to negative results showing bounds on the (in)approximability of various problems were developed. We will concern ourselves here with what we call *list scheduling* problems. These are scheduling problems that can be analyzed using the framework laid out in the previous section, where the input is a list of n jobs and the jobs are processed sequentially. A list schedule is a schedule in which the following is true: if jobs i and j are assigned to the same machine, and if $i < j$ in the input list, then job i precedes job j on the machine. We will focus our attention on those list scheduling problems that

admit pseudo-polynomial time algorithms or admit an FPTAS. Interestingly, many scheduling problems were shown to possess an optimal solution if the jobs are arranged in a list according to some rule. Consequently, a pseudo-polynomial time algorithm or an FPTAS to many list scheduling problems gives a corresponding solution to the original problem where the jobs in the input are not provided in any particular order.

We will follow the standard practice in describing the scheduling problem with the three fields $\alpha|\beta|\gamma$. Here α denotes the machine environment, β describes any constraints specific to the problem, and γ describes the objective function. So $P2|List|C_{max}$ would mean that we are considering a list scheduling problem on 2 parallel machines, and we are interested in minimizing the maximum finishing time (the makespan). We will start with a simple example.

Minimizing makespan on two identical parallel machines

In the scheduling problem $P2|List|C_{max}$, the input is a list of n jobs $J_1, ..., J_n$ with processing times $p_1, ..., p_n$ in $\mathbb{Z}_+$. We would like to schedule the jobs on two identical machines such that the makespan is minimized. The ordering in which the jobs are processed in this problem is unimportant, and a list schedule gives an optimal solution to the problem.

The state vector in any stage k is given by $S = (s_, s_2, s_3) = (z, M_1, M_2)$, where z is the value of the objective function for the state vector, and M_i is the make span on machine i. All components are good monotone, and have the lower-is-better property, and there are no PB components. The mapping $\mathcal{F}_k$ takes as input a state vector in $\mathcal{S}_{k-1}$ and a feasible decision y_k. Here, y_k is simply which machine to schedule job k on, and the cardinality of the decision set is clearly bounded by a polynomial in the input size. Conditions 1, 2, and 3 are satisfied and Theorem 3 gives us the following result, originally described in [4]

Corollary 1. *Branch and δ-Dominate gives an FPTAS for $P2|List|C_{max}$.*

Scheduling To Minimize Weighted Completion Times

In the scheduling problem $P2|List|\sum W_jC_j$, the input is a list of n jobs $J_1, ..., J_n$ with processing times $p_1, ..., p_n$ and weights $w_1, ..., w_n$ associated with each job, in $\mathbb{Z}_+$. We would like to schedule the jobs on two identical machines such that the sum of weighted completion times is minimized. The problem was shown to always have an optimal solution if the jobs are renumbered according to $p_1/w_1 \leq p_2/w_2 \leq ... \leq p_n/w_n$ and processed in a list, and hence an optimal solution to the list problem is an optimal solution to the original problem.

The state vector in any stage k is given by $S = (s_, s_2, s_3) = (z, M_1, M_2)$, where z is the value of the objective function for the state vector and M_i is the makespan on machine i. Like the previous example, all components are good monotone, and have the lower-is-better property, and there are no PB components. The decision y_k is again which machine to schedule job k on. Conditions 1, 2, and 3 are satisfied and we have the following result [4]

Corollary 2. *Branch and δ-Dominate gives an FPTAS for $P2|List|\sum W_jC_j$.*

1.4.2 Scheduling With Dependent Processing Times

We will consider here list scheduling problems with the property that the processing times of the jobs vary according to certain factors. These factors can be the time at which the job starts executing, the machine on which the job is scheduled, the preceding jobs scheduled on the machine, etc. Branch and δ-Dominate gives an FPTAS for many of these problems. We give some examples.

Total Weighted Completion Time With Preceding-Job Dependent Processing Times

In the scheduling problem $P2|List, time - dep|\sum w_jC_j$, we have as input a list of n jobs $J_1, ..., J_n$ with weights $w_1, ..., w_n \in \mathbb{Z}_+$. The processing time p_j of job j on a

machine is a function $p_j : (j, y) \longrightarrow \mathbb{Z}_+$, where y is the last job on the current schedule of the machine. We would like to minimize the total weighted completion time on two identical parallel machines.

At each stage, we can only compare those states that have schedules ending in the same jobs on the machines. The state vector is $(s_1, s_2, s_3, s_4, s_5) = (z, M_1, M_2, l_1, l_2)$, where z is the value of the objective function, M_i is the makespan on machine i, and l_i is the last job scheduled on machine i. s_1, s_2, and s_3 are all good monotone, while s_3 and s_4 are PB. The decision y_k places the job either on machine 1 or machine 2. Conditions 1, 2, and 3 are fulfilled, and we have

Corollary 3. *Branch and δ-Dominate gives an FPTAS for $P2|List, time{-}dep|\sum w_jC_j$.*

For a single machine, one can get strong approximation results for problems with a similar flavor (see [13]).

Total Weighted Completion Time With Starting-Time Dependent Processing Times

In the list scheduling problem $P2|List, time - dep|\sum w_jC_j$, we have as input a list of n jobs $J_1, ..., J_n$ with weights $w_1, ..., w_n \in \mathbb{Z}_+$. For every job j, there is an associated positive integer c_j, and the processing time p_j is equal to c_jt_j, where t_j is the time at which job j starts execution on the machine. We again would like to minimize the total weighted completion time on two identical parallel machines. A job interchange argument shows that renumbering the jobs such that $c_1 \leq c_2 \leq ... \leq c_n$ and processing them in that order gives an optimal solution to the non-list version of the problem.

The state vector at stage k is given by $(s_1, s_2, s_3) = (z, M_1, M_2)$. All components are good monotone having the lower-is-better property. The decision y_k is which machine to schedule job k on. Conditions 1, 2, and 3 are fulfilled, and Theorem 3 gives this result shown before in [14].

Corollary 4. *Branch and δ-Dominate gives an FPTAS for $P2|List, p_j = c_jt_j|\sum w_jC_j$.*

1.4.3 Single Machine Problems

We now discuss two list scheduling problems on one machine.

Scheduling With Rejection

In a scheduling with rejection problem, we are allowed to choose to not schedule (reject) jobs at a certain penalty for each job rejected, and the goal is to minimize some objective criterion, as well as the cumulative penalty for the rejected jobs. More formally, we define the list scheduling problem $1|List|\sum_S w_jC_j + \sum_{\overline{S}} q_j$, where we have n jobs to schedule on one machine, with each job having a processing time p_j, weight w_j, and rejection penalty q_j, and we'd like to choose some subset S of the n jobs to schedule on the machine such that the sum of weighted completion times of the scheduled jobs and the cumulative penalty for the set of rejected jobs (denoted by $\overline{S}$) are minimized. This problem is sometimes also known as scheduling with outsourcing, and the formulation given here is due to Sengupta et al. [15].

To put $1|List|\sum_S w_jC_j + \sum_{\overline{S}} q_j$ into our framework, we define the state vector in state k to be $(s_1, s_2) = (z, M)$, where z is the objective function value at stage k and M is the makespan of the machine. According to the decision y_k, the mapping $\mathcal{F}_k$ schedules the job on the machine, or rejects it. Both components are good monotone with the lower-is-better property, and conditions 1, 2, and 3 are fulfilled. This gives another proof of the following result described in [15]

Corollary 5. *Branch and δ-Dominate gives an FPTAS for* $1|List|\sum_S w_jC_j + \sum_{\overline{S}} q_j$.

Scheduling With Rejection Under Lateness and Outsourcing Constraints

In the scheduling problem $1|List|\sum_S w_jT_j + \sum_{\overline{S}} q_j$, we have as input a list of n jobs, each job in the list is has the following attributes $[p_j, w_j, d_j, l_j, q_j]$, where p_j is the processing time of job j, w_j is its weight, d_j is its due date. The tardiness of job j is defined as $T_j = \max\{0, C_j - d_j\}$, and is constrained to be at most l_j. We have the option of 'rejecting', or not scheduling certain jobs at the expense of incurring a

penalty q_j for rejecting job j. We constraint the number of jobs that we can reject to be at most U. We'd like to choose, under the problem constraints, a subset S of the jobs to schedule on the machine such that we minimize $\sum_S w_j T_j + \sum_{\overline{S}} q_j$, where $\overline{S} = n - S$ is the subset of jobs rejected.

Define the state vector for state k as $S = (s_1, s_2, s_3) = (z, M, u)$, with z the value of the objective function corresponding to that state vector, M being the makespan of the machine, and u the number of jobs rejected so far. Component 2 is a bad monotone, since approximating it might violate the hard constraint imposed on the tardiness. Component 3 is PB, with any values for $s_3 > U$ being infeasible. The mapping $\mathcal{F}_k$ takes as input one of two decisions y_k, either schedule job k on the machine, or reject it. Conditions 1, 2, and 3 are fulfilled, and we have a result that was proved by Sengupta in [16]

Corollary 6. *Branch and δ-Dominate gives an FPTAS for* $1|List|\sum_S w_j T_j + \sum_{\overline{S}} q_j$.

1.4.4 Maximization Problems and Polynomial Objective Functions

So far, all the problems we have concerned ourselves with were minimization problems, where all the monotone components also had the lower-is-better property. All our objective functions have been linear as well. We give here an example of a maximization problem and a problem were the objective function is not linear, but polynomial.

The Knapsack Problem

In the knapsack problem ([17],[3],[18]) the input consists of n elements, with element j being a pair of integers (v_j, w_j) in $\mathbb{Z}_+$, and an integer W in $\mathbb{Z}_+$ as well. v_j is the value of item j, while w_j is its weight. We need to select a subset of the items such that the total value is maximized while the total weight remains less that or equal W.

To put Knapsack into our framework, we define the state vector in state k as $(s_1, s_2) = (v, w)$, where the first component is the total value of the items and the second component is the total weight of the items in the knapsack at stage k. Here, we would like to maximize s_1 without violating the constraint on s_2. Component s_1 is good monotone, while s_2 violates the first condition of goodness and thus is bad and cannot be approximated, as it may prevent us from including another item of an arbitrarily high value in the knapsack. The mapping $\mathcal{F}_k$ can take one of two possible decisions y_k as input: Either put the item in the knapsack or discard it. Conditions 1, 2, and 3 are fulfilled, and the following corollary follows from Theorem 3.

Corollary 7. *Branch and δ-Dominate gives an FPTAS for the knapsack problem.*

Minimizing the Sum of Non Linear Functions of Completion Times on Two Machines

In the problem $P2|List|\sum_{j\in M^1} C_j^2 + \sum_{j\in M^2} C_j^3$, we have as input a list of n jobs $J_1, ..., J_n$ with processing times $p_1, ..., p_n$ in $\mathbb{Z}_+$. We'd like to schedule the jobs on two identical machines such that the aggregate of the sum of the squared completion times of the jobs on machine 1 and the sum of the cubed completion times of the jobs on machine 2 is minimized, subject to the constraints that both machines have exactly $n/2$ jobs scheduled on each of them, and that at each stage k, the makespan of the jobs scheduled on machine 1 is at most b_k.

The state vector in stage k is $S = (s_1, s_2, s_3, s_4) = (z, M_1, M_2, u)$, where u is the number of jobs scheduled on machine 1, and z is the value of the objective function corresponding to that state vector. Components s_1, s_2, and s_3 are monotone, and all have the lower-is-better property while s_4 is PB. Furthermore, Component s_2 has a hard constraint placed on its value at each stage k, violating the first goodness condition. The mapping $\mathcal{F}_k$ takes any state in $\mathcal{S}_{k-1}$ and the decision y_k, and outputs two states in $\mathcal{S}_k$, corresponding to placing the job either on machine 1 or on machine 2. Here, the objective function is a polynomial of degree 3, and we have $\beta_1 = \beta_{max} = 3$

fulfilling goodness condition c for s_1. Furthermore, Conditions 1, 2, and 3 are satisfied, and we have by Theorem 3

Corollary 8. *Branch and δ-Dominate gives an FPTAS for $P2|List|\sum_{j\in M^1} C_j^2 + \sum_{j\in M^2} C_j^3$.*

It is interesting to note that so far, we have not encountered any problem where the value of β_i was not equal to 1 if $i \neq 1$. We have assumed that the first component in a state vector is always the value of the objective function of the corresponding solution, and so all components outside of the objective function have not had any values for β other than 1 in any of our examples. We will revisit this point in the next section.

1.5 Comparisons With Woeginger

In this section, we compare and contrast Woeginger's results on dynamic programming benevolence [5] with Branch and δ-Dominate. A quick review of Woeginger's methods would help in identifying similarities and differences between the two approaches. Woeginger's work considers a generic optimization problem GENE, which can be formulated as a simple dynamic program DP. If DP fulfills certain arithmetical conditions, then GENE is called DP-Benevolent, and admits an FPTAS. DP relies on the trimming the state space technique by merging states that are 'close' to each other. This closeness is decided based on the concept of $[D, \Delta]$-closeness between states. For each problem GENE, there is a degree vector $D = [d_1, ..., d_\alpha] \in \mathbb{N}^\alpha$, that only depends on GENE and on the DP formulation, but not on any specific instance of GENE. For a real number $\Delta > 1$ and two state vectors $S = (s_1, ..., s_\alpha)$, and $S' = (s'_1, ..., s'_\alpha) \in \mathbb{N}^\alpha$, we say that S is $[D, \Delta]$ to S' if for $l = 1, ..., \alpha$, we have

$$\Delta^{-d_l} \cdot s_l \leq s'_l \leq \Delta^{d_l} \cdot s_l$$

Before proceeding to the benevolence conditions given by Woeginger, we need to introduce two sets of functions $\mathcal{F}$ and $\mathcal{H}$, and a function G. First note that the input

to GENE is given by n vectors $X_1, ...X_n \in \mathbb{N}^\beta$. The set $\mathcal{F}$ is a finite set of mappings $\mathbb{N}^\alpha \times \mathbb{N}^\beta \to \mathbb{N}^\alpha$. The set $\mathcal{H}$ is a finite set of mappings $\mathbb{N}^\alpha \times \mathbb{N}^\beta \to \mathbb{R}$. For every function $F \in \mathcal{F}$, there is a corresponding mapping $H_F \in \mathcal{H}$. The state space $\mathcal{S}_k$ is obtained from $\mathcal{S}_{k-1}$ via $\mathcal{S}_k = \{F(X_k, S) : F \in \mathcal{F}, S \in \mathcal{S}_{k-1}, H_F(X_k, S) \leq 0\}$. The function $G : \mathbb{N}^\alpha \to \mathbb{N}$ is a non-negative function which gives the optimal objective value $OPT(I)$ to an instance I of GENE according to $OPT(I) = \min\{G(S) : S \in \mathcal{S}_n\}$.

The concept of dominance in DP is captured through two binary relations $\preceq_{dom}$ and $\preceq_{qua}$ on $\mathbb{N}^\alpha$. The dominance relation $\preceq_{dom}$ is a partial order on $\mathbb{N}^\alpha$, while $\preceq_{qua}$ is a quasi-linear order on $\mathbb{N}^\alpha$. A relation on a set Z is a quasi-linear order if its reflexive and transitive, and if any two elements of Z are comparable. A relation $\preceq_{qua}$ is any extension of $\preceq_{dom}$, meaning that for two states S and $S' \in \mathbb{N}^\alpha$, $S \preceq_{dom} S'$ implies $S \preceq_{qua} S'$. We are now ready to give the benevolence conditions for DP.

Condition 1. *For any $\Delta > 1$, for any $F \in \mathcal{F}$, for any $X \in \mathbb{N}^\beta$, and for any $S, S' \in \mathbb{N}^\alpha$, the following holds:*

i) If S is $[D, \Delta]$-close to S' and if $S \preceq_{qua} S'$, then (a) $F(X, S) \preceq_{qua} F(X, S')$ holds and $F(X, S)$ is $[D, \Delta]$-close to $F(X, S')$, or (b) $F(X, S) \preceq_{dom} F(X, S')$ holds.

ii) If $S \preceq_{dom} S'$, then $F(X, S) \preceq_{dom} F(X, S')$.

Condition 2. *For any $\Delta > 1$, for any $H \in \mathcal{H}$, for any $X \in \mathbb{N}^\beta$, and for any $S, S' \in \mathbb{N}^\alpha$, the following holds:*

i) If S is $[D, \Delta]$-close to S' and if $S \preceq_{qua} S'$, then $H(X, S') \leq H(X, S)$.

ii) If $S \preceq_{dom} S'$, then $H(X, S') \leq H(X, S)$.

Condition 3. *The following are conditions on the function G.*

i) There exists an integer $g \geq 0$ whose value depends only on G and D such that for any $\Delta \geq 1$ and for any $S, S' \in \mathbb{N}^\alpha$, the following property holds: If S is $[D, \Delta]$-close to S' and if $S \preceq_{qua} S'$, then $G(S') \leq \Delta^g \cdot G(S)$.

ii) If $S \preceq_{dom} S'$*, then* $G(S') \leq G(S)$.

In addition to these conditions, a set of technical conditions regarding the size of the input and the computation complexity of the functions involved is also required. The following is the main result of Woeginger's paper, which states that a DP optimization problem GENE is called DP-benevolent iff there exist a partial order $\preceq_{dom}$, a quasi-linear order $\preceq_{qua}$, and a degree vector D such that its dynamic programming formulation fulfills the above conditions

Theorem 4. *If an optimization problem GENE is DP-benevolent, then it has an FPTAS.*

We now compare DP-benevolence to Branch and δ-Dominate. The first obvious difference between the two methods lies in how the state vectors are described. There in no distinction between state components in Woeginger's framework, and the concept of (good and bad) monotone or PB components is absent. This distinction is the cornerstone that Branch and δ-Dominate relies on to classify whether a problem admits an FPTAS or not.

We noted earlier that both approaches make use of slightly altered versions of the trimming the state space technique. While it can be argued that Woeginger's concepts of $[D, \Delta]$-closeness, and the binary relationships $\preceq_{dom}$ and $\preceq_{qua}$ are more general and can offer more flexibility than the simpler concepts we employ here, we know of no problems in the literature that can be captured using this added generality, while at the same time being elusive to δ-domination.

It is interesting to note that, out of the 11 problems considered by Woeginger in [5], there is not a single one of them where the component d_l in the degree vector takes a value that is more than 1, unless component l in the state vector happens to be the objective function (i.e. all components of the state vector, with the possible exception of the objective function, are always linear.) This is consistent with our earlier observation in the previous section.

The problems in Woeginger are divided into two classes. The extremely benevolent (ex-benevolent) DP problems, and the critical-condition benevolent (cc-benevolent) DP problems. Ex-benevolent problems can be defined as those problems with $H \equiv 0$ for all $H \in \mathcal{H}$, and where $\preceq_{dom}$ is the trivial relation on $\mathbb{N}^{\alpha}$, and $\preceq_{qua}$ is the universal relation on $\mathbb{N}^{\alpha}$. In the cc-benevolent problems, there is a coordinate in the state vector that is referred to as the critical coordinate, and the relationship $\preceq_{qua}$ is the quasi-linear order $\preceq_{cc}$ on the states, where $S \preceq_{cc} S'$ if the critical coordinate of S' is less than or equal to that of S. Woeginger gives various lemmas to determine if the benevolence conditions, or variants of them, are satisfied for these two classes of problems. We don't have to make these distinctions here, as our goodness conditions capture both cases and all the problems are treated the same way under Theorem 3.

Two things included in this chapter that are not in Woeginger's work are the inclusion of multi-criteria optimization and handling a subset of problems where the cardinality of the decision set at each stage can be exponential. Woeginger's work has been extended to multi-criteria by Angel, Bampis, and Kononov [19]. In what follows, we show that the extension of the approach we presented so far to the case of multiple criteria is natural and intuitive.

1.6 Multi - Criteria problems

In a multi-criteria optimization problem, solutions are evaluated with respect to more than one cost criteria. Typically, a solution to such a problem is given by the Pareto curve, which gives the trade-off between the various criteria being optimized. The Paerto curve is basically the set of all undominated feasible objective vectors, and can be defined more formally as follows. Given an instance I of a multi-criteria optimization problem in k objective functions $f_i, i = 1, ..., k$; the Pareto curve $P(I)$ is the set of all k-vectors of values such that for each vector $v \in P(I)$, *(1)* there is a feasible solution x such that $f_i(x) = v_i$ for all i, and *(2)* there is no feasible solution x' such that $f_i(x') \geq v_i$ for all i, with strict inequality for some i.

As expected, the Parteo curve is hard to compute, and we again have to settle for approximate solutions. For an accuracy parameter $\epsilon > 0$, we define an ϵ-*approximate* Pareto curve, denoted $P_\epsilon(I)$, as the set of solutions such that for some solution $x \in P_\epsilon(I)$, there is no other solution x' such that $f_i(x') \geq (1+\epsilon)f_i(x)$, for all i. Papadimitriou and Yannakakis showed that an ϵ-approximate Pareto curve always exists [6], and that it contains a number of solutions that is polynomial in $|I|$ and $1/\epsilon$, but exponential in the number of objectives.

Branch and δ-Dominate extends naturally to handle multi-criteria optimization, whereas the various objectives to be optimized are just written down in the state vector as components that can be approximated. The definitions of bad and good monotone components still apply, and help us in determining whether the ϵ-approximate Pareto curve for a multi-criteria *LO* problem can be efficiently computed via this method. If the state vector component corresponding to one of the criteria is bad, then this criteria is referred to as a bad criteria. We have the following theorem

Theorem 5. *IF Branch and δ-Dominate gives an FPTAS for some single-criterion combinatorial optimization problem, then it also gives an FPTAS for computing an ϵ-approximate Pareto curve for a multi-criteria version of that problem, provided that at most only one criteria is bad.*

Proof. Because the various citeria are simply represented as components in the state vector, the proof proceeds exactly like the single criterion case, and the running time is polynomial in the size of the problem and in $1/\epsilon$. Since we are interested in more than one objective, the running time will be exponential in the number of criteria, as observed by Papadimitriou and Yannakakis. Consider the optimal Pareto curve $P(I)$ for the problem obtained by an optimal algorithm. At the end of stage n, and for each state $S \in P(I)$, there exists a state S' returned by Branch and δ-Dominate, with at least one of the objective criteria being at most $(1+\epsilon)$ or $(1-\epsilon)$ away from its optimal value in S. This is a straightforward consequence Theorem 3. A maximal undominated subset of these state vectors forms the ϵ-approximate Pareto set, $P_\epsilon(I)$. □

Let us illustrate this result with an example. Interestingly, the solution method is exactly the same as the single criterion problems, showcasing the simplicity and power of the approach.

1.6.1 Minimizing Makespan and Sum of Cubed Completion Times on Two Machines

In the list scheduling problem $P2|List|C_{max}, \sum C_j^3$, we're given as input a list of n jobs $J_1, ..., J_n$ with processing times $p_1, ..., p_n \in \mathbb{Z}_+$, and we'd like to schedule them on two identical machines such that the makespan and the sum of cubed completion times are minimized. This is a bicriteria optimization problem, and the solution we're looking for is an ϵ-approximate Pareto curve giving the trade-off between the two objectives.

In stage k, we have $S = (s_1, s_2, s_3, s_4) = (C_{max}, \sum C_j^3, M_1, M_2)$, and all the components are good monotone. The decision y_k at each stage is simply which machine to place the current job on. Since Branch and δ-Dominate gives an FPTAS for the single criterion version of this problem if we consider just one of the two objectives, and since all components fulfill the goodness conditions with $\beta_{max} = 3$, Theorem 5 tells us that we can compute an ϵ-approximate Pareto set, leading to the following result, which also appears in [19]

Corollary 9. *Branch and δ-Dominate gives an FPTAS for constructing an ϵ-approximate Pareto curve for the bicriteria list scheduling problem $P2|List|C_{max}, \sum C_j^3$.*

1.6.2 Problems with 'Crashing'

We consider now a special case of problems where the number of decisions we can make per stage is exponential. On first glance, it seems hopeless to construct an FPTAS for these problems using Branch and δ-Dominate. However, it turns out that this is not an issue for the certain class of problems that we consider here, and an FPTAS can indeed be constructed with little modification to Branch and δ-Dominate.

Let us first introduce the concept of 'crashing'; we are given an initial budget endowment that we are allowed to use to decrease, or 'crash' some values in the input data, probably leading to another decrease in the values of some monotone components in the problem's state vector. An example would be decreasing the processing time of a job on a machine in a scheduling problem or reducing the weight of an item in a knapsack problem. One can think of crashing as using budget to buy more resources so that a job executes faster (i.e. has shorter processing time.) Now, besides the usual decisions we were making in the previous problems, like which machine to schedule a job on, etc., we have another set of decisions to make, namely, which jobs to crash, how much to crash a particular job, etc. It is clear that for some values of the budget, the number of decisions that can be made per stage is exponential, reflecting a trade-off between budget usage and processing time. We examine this more formally in the next section.

List Scheduling with Crashing

We consider list scheduling problems with the property that the processing times of jobs can be shortened by using up some budget of an initial budget endowment B. The processing time of job j becomes a function $p_j(b)$, where b is the amount of budget used to crash the job. We assume that for two values b and b', we have $p_j(b) \geq p_j(b')$ if $b' \geq b$. We need to modify Branch and δ-Dominate in order to account for this new situation. Again, we revisit the makespan minimization problem to illustrate this modification in the context of an example.

Minimizing Makespan on 2 Machines with Budget Constraints

In the problem $P2|List, B|C_{max}$, we're given as input a list of n jobs $J_1, ..., J_n$ with processing times $p_1, ..., p_n$ and an integer B in $\mathbb{Z}_+$. The processing times of the jobs are their non-crashed values (i.e. $p_i(0)$). We'd like to schedule the jobs, with crashing, on two identical machines such that the completion time of the last job finishing is minimized.

The state vector here comprises three components, $S = (M_1, M_2, b)$, corresponding to the makespans on the two machines and the budget used up by the solution corresponding to that state vector. Note that b is constrained to take values that are at most B. The decisions to be made at stage i now are: a) Which machine to place job i on, and b) How much to crash job i. A problem arises now with the value of B, which can be exponential in the problem size, and thus trying all possible values of b for crashing a job is not an option. To get around this difficulty, the budget allocated to job i is determined through a binary decision process as follows. If the budget can take values in $[l, u]$, with $l = 0$ and $u = B$ initially, we evaluate the processing times $p_i(b)$ when $b = l$ and when $b = u$, for all state vectors. If $p_i(l)$ δ-dominates $p_i(u)$ for a state, we assign the budget l and the processing time $p_i(l)$ to that state, and we call this a closed interval. Otherwise, we branch again by having two new intervals, $[l, \frac{(l+u)}{2}]$ and $[\frac{(l+u)}{2} + 1, u]$, and recursively repeat the process on each of the intervals. We stop branching on an interval and close it whenever the condition is satisfied, and we assign the lower budget value, together with the corresponding processing time of the crashed job to the state.

This is the general method of budget allocation to any list scheduling problem that involves crashing. The following is a general result that applies to all these types of problems.

Lemma 6. *The number of states resulting from the binary decision process on the budget is polynomial in the input size of the problem and in* $1/\delta$.

Proof. Denote by P the largest processing time in the input data for an instance. Suppose further that P is greater than B. Let us divide the interval $[0, P]$ into $log_{1+\delta}(P)$ strips or boxes. Suppose we have decided to place the current job on machine i and are considering how much to crash it by employing the procedure described above. We notice that any open interval $[b_1, b_2]$ is such that $p_i(b_1)$ and $p_i(b_2)$ fall in different boxes, because otherwise the interval $[b_1, b_2]$ will be closed as $p_i(b_1) \leq (1+\delta)p_i(b_2)$. Consequently, there can be as many open intervals as there are boxes, and that number is at most $log_{1+\delta}(P)$. This in turn is equal to $\ln(P/\ln(1+\delta)) \leq$

$\lceil (1 + 1/\delta) \ln(P) \rceil$ (from the Taylor series expansion of $\ln(1 + \delta)$). The right hand side term in the last inequality is equal to $\lceil (1 + 1/\delta) \log(P)/\log 2 \rceil = O(\frac{1}{\delta} \log(P))$.

On the other hand, suppose that B is bigger than P, and suppose that we keep branching on one interval until two neighboring integers fall in their own boxes. This would require that we do $O(\log(B))$ branchings, which can potentially be bigger than $O(\frac{1}{\delta} \log(P))$. The running time is then given by $O(\max\{\log B, \frac{1}{\delta} \log(P)\}) \leq O(\frac{1}{\delta} \log \max\{B, U\})$, proving the lemma. □

This result tells us that, even though we have an exponential number of decisions at every stage, using the branching process reduces the number of decisions in every stage to a polynomial in the input size and $1/\epsilon$, and our main theorems still apply. From the lemma, and from theorem 2, Branch and δ-Dominate will take time polynomial in the size of the input and in $1/\epsilon$ to terminate. At stage n, we will have a subset of nodes that represent the trade-off between the budget used and the corresponding makespan.

Minimizing Total Weighted Completion Time on 2 Machines with Budget Constraints

In the list scheduling problem $P2|List, B|\sum w_j C_j$, we have as input a list of n jobs $J_1, ..., J_n$ with processing times $p_1, ..., p_n$ and weights $w_1, ..., w_n$, in addition to a budget B, all in Z_+. We'd like to schedule the jobs on two identical machines such that the total weighted completion time of the jobs is minimized subject to the budget constraint.

The state vector here in stage k is $(s_1, s_2, s_3, s_4) = (z, M_1, M_2, b)$, where z is the value of the objective function, M_i is the makespan on machine i, and b is the budget used so far. The decisions at each stage are which machine to place the job on and how much to crash it. Components 1,2, and 3 are strongly monotone and good. Component 4 is bad as we operate with a constrained budget B. Since we have just

shown that the cardinality of the decision set at every stage is still bounded by a polynomial in the input size, then using Theorem 3 gives us the following result

Corollary 10. *Branch and δ-Dominate gives an FPTAS for $P2|List, B|\sum w_j C_j$*

Of course, this analysis will extend to any problem where we are allowed to use the budget to manipulate the values in the input.

1.7 Summary Of Results for Scheduling Problems

We have presented existence conditions for FPTASes and pseudo-polynomial time algorithms for single and multiple criteria list scheduling problems. In this section, we tie the approach presented in this chapter to various results in the scheduling literature.

Before proceeding, we note that any results that are applicable to uniform and parallel machine environments also hold true for independent and parallel machine environments as well, as per the following theorem. Note that independent machines only differ from uniform machines in that the processing time of the same job can be different from one machine to the next.

Theorem 7. *Good and bad monotone components retain their status regardless of the machine environment.*

Proof. Consider a problem in a parallel/uniform machine environment and a state vector S_k having at most one bad component. From our earlier results, this problem possesses an FPTAS. Now, replace the machines in that environment with the same number of non-identical, parallel machines. Let us examine the three conditions in Definition 7. For the first condition, if component i in a state vector could be replaced by another non-negative integer value and the resulting state vector is feasible, then this implies that component i is not constrained, and this property holds whether the machines are independent or uniform (as an example, if component i should not

exceed a value m, then it will be bad regardless of the machine environment.) Since all the input to our problems (processing times, weights, etc.) is in $\mathbb{Z}_+$, the second goodness property will hold as well for a component that was good in a uniform environment, as the value of s_i can only increase in an independent environment when adding a new job to a machine if that was the case in a uniform environment as well (otherwise the component would have been bad in the uniform environment; remember that the only thing that changes is the processing times of the jobs.) Finally, note that the third goodness property depends on $\mathcal{F}_k$ and y_k, and not on the machine. One can say that $\mathcal{F}_k$ takes as input a processing time and the decision y_k. If condition 3 of goodness was satisfied for a certain component for a certain value of p_j, then it will still be satisfied for another value of p_j, and hence the machine environment does not affect this condition as well. The arguments from the non-uniform to uniform environment are similar.

□

1.7.1 Problems With an FPTAS

We consider some problems that admit an FPTAS.

Problems On Two (or more) Machines

Consider scheduling problems of the form $P2|List|minc()$, where $c()$ is an objective function like $\min \sum_j (w_j C_j)^q$. Let us analyze the state vector for various constraints. All these problems posses an FPTAS as per theorem 3.

- Consider an upper bound $U(k)_i$ on the number of jobs that can be processed on machine i at stage k. the state vector S_k is $(s_1, s_2, s_3, s_4, s_5) = (z, M_1, M_2, u_1, u_2)$, with u_i being the number of jobs processed on machine i.

- Consider a lower bound L_i on the number of jobs that can be processed on machine i at stage k. the state vector S_k is $(s_1, s_2, s_3, s_4, s_5) = (z, M_1, M_2, u_1, u_2)$, with u_i being the number of jobs processed on machine i.

- Consider allowing rejection, with a bound O on the number of jobs that can be rejected. the state vector S_k is $(s_1, s_2, s_3, s_4) = (z, M_1, M_2, o)$, with o being the number of jobs rejected so far for this state vector.

- Consider an upper or lower bound constraint on a monotone component, like M_i, the state vector will include a bad monotone component corresponding to the constrained term.

We can also consider variations on the problem where the processing times of the jobs vary according to some criterion. For example

- The processing time p_j is a function of the job immediately preceding job j on the machine. The state vector will be of the form $S_k = (s_1, s_2, s_3, s_4, s_5)) = (z, M_1, M_2, l_1, l_2)$, where l_i, is the last job on machine i.

- The processing time p_j is a smooth function of the time at which job j starts executing. The state vector will be of the form $S_k = (s_1, s_2, s_3) = (z, M_1, M_2)$.

Comments. There is still an FPTAS for all these problems for any fixed number m of machines. Also, any combinations of these constraints will still give us a problem with an FPTAS. There is also still an FPTAS regardless of the machine environment as per Theorem 7.

Problems On a Single Machine

Consider scheduling problems of the form $1|List|c(.)$. Examples for $c(.)$ can be $\min \sum_j (w_j C_j)^q$, or $\sum w_j U_j$, or $\sum w_j T_j$. Let us analyze the state vector for various monotone constraints. All these variants have an FPTAS as long as the number of bad monotone components is at most one.

- Consider the problem where each job j has a strict due date d_j. The state vector is $S_k = (s_1, s_2) = (z, M)$, component 1 is bad monotone.

- Consider the problem where each job j has a due date d_j and a maximum lateness L_j. The state vector is $S_k = (s_1, s_2) = (z, M)$, component 1 is bad monotone.

Comments. Combining any one of these constraints with any of the constraints in the previous section still results in a FPTAS. There is no FPTAS if the numbers of the machines is more than one. Other existing results in the literature that will fit in our framework are [8], [7], and [13].

1.8 Conclusions

In this chapter, we have presented Branch and δ-Dominate, a powerful framework for quickly determining the existence or lack thereof for list optimization problems. We have compared and contrasted our approach with that of Woeginger, and argued that ours is simpler and easier to work with. We have also argued that while Woeginger's approach seems to be more encompassing, we were not able to find evidence that there is a wide variety of problems that benefit from the added complexity.

We showed how our approach can be used to give FPTASes for a wide range of list scheduling problems. We extended the approach to handle multi-criteria optimization and showed that in the context of branch and dominance, the extension is a natural and intuitive one. We also presented a special class of problems where despite the exponential cardinality of the decision set at each stage, we were able to still get an FPTAS through our binary branching procedure presented in Section 1.6.2. In the last section, we summarized some results for scheduling problems.

In the next chapter, we study the stochastic lot sizing problem and give an FPTAS under certain assumptions. The problem is fairly more complicated than any of the problems we considered here, and though a direct application of branch and dominance or the concept of good and bad components does not apply, we still employ some of the techniques we used in this chapter to derive our results.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Stochastic Lot Sizing

2.1 Introduction

Inventory management and lot sizing problems have always been the central area of research in the supply chain and operations management literature. This problem arises in a multitude of domains and has many practical applications (for example, [20] and [21]). A significant portion of the research conducted in this area has been dedicated to finding optimal control policies for problems with stochastic and uncertain demands. Many of these policies turned out to have surprisingly simple characterizations, like for instance the state-dependent base-stock policy [22], where in each period there exists an optimal target base-stock level that is determined only by the given conditional distribution at that period on future demands, but is independent of the inventory level at the beginning of the period. The optimal policy aims to keep the inventory level at every period as close as possible to the target base-stock level. That is, it orders up the target level whenever the inventory level at the beginning of the period is below that level, and orders nothing otherwise. A slightly more complicated class of policies is the class of state-dependent (s, S) policies [23], where in each period there are lower and upper thresholds that are determined only by the conditional future demand distribution in that period. The optimal policy places an order in a period if the the inventory level at the beginning of the period is below the lower threshold. This order is placed to bring the inventory up to the

upper threshold.

Perhaps a major part of the reason these policies turned out to be as simple is that the dominant framework in which these problems were usually formulated was mostly straightforward dynamic programming recursions. Unfortunately, on the flip side, these policies also inherited the infamous 'curse of dimensionality' that plague dynamic programming as well, making it almost impossible to find computationally tractable algorithms for computing the optimal policies. Because of this intractability, many researchers have attempted to construct computationally efficient heuristics for these problems (see, for example [24]. But no attempts have been done to analyze the worst-case performance ratio of these heuristics. Perhaps the most famous class of these heuristics is what is known as myopic policies [25],[22], where in each period we attempt to minimize the expected cost for that period, ignoring the effect on the cost of future periods. While myopic policies are attractive because of their simplicity, they can perform arbitrarily badly in some situations. Trying to approximate the huge dynamic programs through a Markov chains approach was another approach taken in [26].

Recently though, there has been a surge in interest in finding algorithms that, while perhaps not optimal, are efficiently implementable and guarantee a performance that is comparable to the optimal solution. Most notable here is the work of Levi, Pal, Roundy, and Shmoys on approximation algorithms for stochastic inventory control models [27]. In their work, they give a 3-approximation algorithm for the stochastic lot-sizing problem. An α-approximation algorithm for a minimization problem guarantees its output to be no more than α times the optimal solution, regardless of the instance. According to the authors, this has been the first contribution in the literature that moves from heuristics with no formal guarantees on the quality of the solution into providing a computationally tractable procedure with a worst case performance guarantee. In their approach, the authors stray from the conventional dynamic programming paradigm and develop more complicated analytical techniques to reach their results.

In this chapter, we strike a middle ground between the two approaches, in the sense

that we wish to find computationally efficient and provably good solutions, but we do so in a dynamic programming framework. We investigate the stochastic lot-sizing problem, and show that under some conditions, one can give a fully-polynomial time approximation scheme (FPTAS) to obtain ϵ-approximate solutions. FPTASes are considered by many to be the 'gold standard' in approximation algorithms, whereby one is able to get as close as they wish to the optimal solution at the expense of more running time. The running time is polynomial in the size of the problem and also depends polynomially on the inverse of the accuracy parameter ϵ. We note that, while our approximability results are stronger than those of Levi, Roundy, and Shmoys, they are so because we employ a set of more restrictive assumptions.

The remainder of this chapter is structured as follows. Section 2.2 explains the model we use and states our assumptions. Section 2.3 gives pseudo-polynomial time algorithms for the problem. These algorithms are the main building blocks of FPTASes, and form the basis of obtaining our main result. Section 2.4 introduces a couple of general techniques for computing approximate functions, which are later used to derive the ϵ-approximations. Section 2.5 concludes the chapter.

2.2 Models And Assumptions

This section introduces the basic notation, models, and assumptions used throughout the rest of the chapter. Lot sizing and inventory management problems exhibit a richness in the amount of interesting variations that stem from the central problem. We have tried to make our general model as fairly encompassing as we could under the restriction that we are interested in an FPTAS.

In the single-item stochastic lot-sizing problem, we have n planning periods. In each period j, $j = 1, ..., n$, there is a stochastic, integer-valued demand d_j for a single product, that comes from a known distribution $F_j(.)$. Thus, the probability that the demand in period j is equal to k is given by $F_j(k) - F_j(k-1)$ and is denoted by P_{jk}. We assume that the demand distribution functions are known for each period, and that they are independent of each other; the actual demand value is revealed at

the end of the period. In each period j, we have the option to produce a quantity x_j in order to fulfill current as well as future demand. If we decide to produce in a particular period j, we incur a fixed cost K_j plus a variable cost $c_j(x_j)$, where $c_j(.)$ is a monotone non-decreasing function. Inventory can either be held from one period to the next, or it can be disposed of. There is a holding cost $h_j(.)$ to hold inventory from period j to period $j+1$, where $h_j(.)$ is again a monotone non-decreasing function. We will assume that $c_j(0)$ and $h_j(0)$ are equal to zero. Disposal of inventory can be done at zero cost at any time, i.e. the salvage value at any period is zero. For simplicity, and whenever no ambiguity can arise, we will drop the subscript j.

Demand can be fully or partially fulfilled. If we denote by I_j the inventory at the end of period j and the beginning of period $j+1$, then partial fulfillment of demand in period $j+1$ occurs when $I_j + x_{j+1} < d_{j+1}$, i.e. the sum of the amount of inventory at the beginning of the period and the amount produced is less than the actual demand for that period (demand for the period is revealed after production decisions are made). Regarding partial demand fulfillment, we will consider two models. The first assumes that any demand not fulfilled in a particular period j is immediately lost at a cost. The cost is represented by a function $q_j(s)$ which is monotone non-decreasing in the amount of lost demand s. Again, we assume that $q_j(0) = 0$. The second model does not allow demand to be lost, but rather, it is backlogged –again at a cost– and fulfilled at a later period. Finally, we assume that all cost functions are bounded above, and that if the argument to a function is not zero, then the value of that function is bounded below by 1. The objective of the stochastic lot-sizing problem is to minimize the overall cost.

Pseudo-polynomial time (PPT) algorithms are the basic building blocks for FPTASes. For lot-sizing problems, PPT algorithms are usually dynamic programs with prohibitive running times. Understanding the structure and inner workings of these dynamic programs is key to developing an FPTAS. For this reason, we discuss the dynamic programming recursions for the our model in the next section. We do not have to impose the free disposal assumption to obtain PPT algorithms, but we will need it later for the approximation schemes. We also assume the existence of an

oracle that evaluates the functions $h(.)$, $q(.)$, and $c(.)$ in $O(1)$ time for any particular argument.

2.3 Pseudo-Polynomial Time Algorithms

We start by giving the notation used in the rest of this section. We denote by $g_j(I)$ the optimum expected cost starting in period j with an inventory of I, so according to our convention, I is just equal to I_{j-1} for stage j, with I_o being defined to be equal to zero. Thus, our goal is to calculate the quantity $g_1(I_0) = g_1(0)$. For period j, let u be the inventory level *after* demand has been observed in that period, and let u^+ and u^- denote $\max\{0, u\}$ and $\max\{0, -u\}$, respectively. Let $r_j(u)$ be defined as $\min\{r_j(u-1), q_j(u^-)+h_j(u^+)+g_{j+1}(u^+)\}$. The term $r_j(u)$ captures all expected costs in period j given that the inventory level is at u. The term $r_j(u-1)$ is included in the min expression because of our free disposal assumption: If $r_j(u)$ has more expected cost than $r_j(u-1)$, then we just dispose of one unit of inventory and assign the cost of $r_j(u-1)$ to $r_j(u)$.

Let us now consider the last period, and again utilize the free disposal assumption. We can see that $r_n(u) = \min\{r_n(u-1), q_n(u^-)\}$ because the term $h_n(u^+) + g_{n+1}(u^+)$ is equal to zero. We can then write $g_n(I)$ as the following program that minimizes the objective function over all possible values for x, the quantity produced, for a particular starting value for the inventory, I. The summation in the second term is taken over all possible demand values for that period.

$$\begin{aligned} \min \quad c_n(x) \quad &+ \quad \sum_k P_{nk} \times q_n(s_k) \\ st \quad s_k \quad &= \quad \max(0, k - x - I) \\ x \quad &\in \quad \mathbb{Z}^+ \end{aligned} \tag{2.1}$$

And in general, for period j, we write $g_j(I)$ as

$$\begin{aligned} \min \quad c_j(x_j) \quad &+ \quad \sum_k P_{jk} \times r_j(I + x - k) \\ x \quad &\in \quad \mathbb{Z}^+ \end{aligned} \tag{2.2}$$

For period j, let z denote the combined inventory resulting from the inventory on hand at the beginning of the current period plus the amount produced in that period, so that $z = I_{j-1} + x_j$. We will denote the second term in the objective function in (2.2) above by $Y_j(z)$, the expected cost *after* a production decision has been made in period j, so that $Y_j(z) = \sum_k P_{jk} \times r_j(z - k)$. We let D^* be equal to the sum of the maximum demand values in all periods, and let d_j be the number of possible different demands in period j. We designate $\max\{d_j : j = 1, ..., n\}$ by d^*. Therefore in (2.2) above, x will be further restricted to take values between 0 and D^*.

Proposition 1. *The time needed to solve the single-item stochastic lot-sizing problem to optimality is* $O(nD^{*^2})$.

Proof. Consider period j. To evaluate $g_j(I)$ for a particular value of I and x (i.e. for a fixed z), we need to consider all possible values for the demand k. As mentioned earlier, the number of these different values is d_j, and this can be at most d^*. Thus for a fixed z we need $O(d^*)$ steps to compute $Y_j(z)$. To compute $Y_j(z)$ for all values of z requires $O(d^* D^*)$ steps. For a fixed I, z varies only as a result of changing x, and to compute $g_j(I)$ we take $O(D^*)$ steps, corresponding to the possible values x can take. To compute $g_j(I)$ for all values of $I \leq D^*$ we therefore take $O(D^{*^2})$. This is because the term D^{*^2} dominates the $d^* D^*$ term required to calculate $Y_j(z)$. Finally, to calculate $g_j(I)$ for all $I \leq D^*$ and all $j = 1, .., n$ we would require $O(n(d^* D^* + D^{*^2})) = O(nD^{*^2})$ steps.

□

Note that the situation when we replace lost sales with backlogging is mathematically identical. The monotone nondecreasing cost function $q_j(s_k)$ for lost sales is replaced by another monotone nondecreasing cost function in the amount of un-

fulfilled demand s_k. Additionally, the starting inventory at every period could be negative, indicating a backlogging situation. In this case the range for the value of inventory at the beginning of a period is in $[-D^*, D^*]$ and the running time remains the same, up to a constant. Since the running time directly depends on d^* and D^* and is polynomial in those quantities, this is indeed a PPT algorithm.

2.3.1 Linear plus fixed costs

We now turn our attention to give an algorithm for the special case when all cost functions are linear and the production cost function for period j is given by

$$c_j(x) = \begin{cases} K_j + \hat{c}_j x, & x = 1, 2, ..., D^*; \\ 0, & x = 0. \end{cases}$$

where K_j is a fixed constant and $\hat{c}_j$ is a cost for producing one unit of inventory. The algorithm relies on calculating $g_j(I)$ using a min-priority queue. This queue is built using a heap data structure, which we will call $Heap(I)$. Each node in $Heap(I)$ has a handle x (we will later refer to this node as node x) and a key that is equal to $c_j(x) + Y_j(I + x)$. The operation $\min(Heap(I))$ simply returns the minimum key value in $Heap(I)$, which is also the same as $g_j(I)$ if the heap contains nodes for all possible values of x. Returning the value of the minimum element in a heap is an $O(1)$-time operation, and thus $\min(Heap(I))$ takes constant time.

How much time does it take to build $Heap(I)$ for some stage j, provided we have calculated $g_{j+1}(.)$? First, we again do the pre-processing step as we did in the previous section, by calculating $Y_j(z)$ for all values of z in $O(d^*D^*)$ time. To build $Heap(0)$, we take $O(D^*)$ time, because we have to start from scratch and create all the nodes in the heap. We have at most D^* nodes, one for each value of x, and so it takes $O(D^*)$ steps to build $Heap(0)$. One would expect however that $g_j(I)$ and $g_j(I+1)$ have very similar structures due to the assumptions on the cost functions. This is indeed the case, and it is where we can exploit the linearity of the cost functions for computing $Heap(I+1)$ from $heap(I)$ in an efficient manner. Let $key(x)$ denote the value of node x in $Heap(I + 1)$, and let $oldkey(x)$ denote the key of node x in $Heap(I)$. Consider

node x, where $1 \leq x < D^*$, then

$$\begin{aligned} key(x) &= c_j(x) + E(Y_j(I + x + 1)) \\ &= c_j(x + 1) - \hat{c}_j + E(Y_j(I + 1 + x)) \\ &= oldkey(x + 1) - \hat{c}_j \end{aligned}$$

Thus, with the exception of $x = 0$ and $x = D^*$, any node x in $Heap(I + 1)$ has a key that is the same as the key of node $x + 1$ in $Heap(I)$, minus $\hat{c}_j$. Note that there is no need to calculate the keys to those nodes in $Heap(I + 1)$ corresponding to a total inventory more than D^*. For example, if we are calculating $Heap(1)$, we do not need to consider a value of D^* as one of our nodes, since this will lead to an overall inventory of $D^* + 1$, which is more than the maximum possible demand in all periods. Therefore, we need to only worry about node 0 in $Heap(I + 1)$. This key is simply the same as $key(1)$ in $Heap(I)$ minus $K_j + \hat{c}_j$. Therefore, to create $Heap(I + 1)$ from $Heap(I)$, we delete element 0 from the heap, increase all node indices by 1, subtract $\hat{c}$ from all keys, and then reinsert the new element 0 with key calculated as above. Both deleting and inserting an element from and into a heap can be performed in $O(\log D^*)$ time. Increasing the indices and subtracting can be performed implicitly in $O(1)$ time. Thus, the total time to update the heap at each iteration is $O(\log D^*)$.

Proposition 2. *The time needed to solve the stochastic lot-sizing problem to optimality when the holding and lost sales cost functions are linear and the production cost function is linear plus a fixed constant is* $O(nD^*(d^* + \log D^*))$.

Proof. We have shown that in stage j, the pre-processing step takes $O(d^* D^*)$ steps. Creating $Heap(0)$ takes $O(D^*)$ steps, one for each value of x, and we do D^* updates to build the heaps for each value of $I \leq D^*$. Updating the heap takes $O(\log D^*)$ steps, for a total time of $O(D^* \log D^*)$. Thus the total time needed to compute $g_j(I)$ for all values of I is $O(d^* D^* + D^* \log D^*)$. Computing $g_j(I)$ for all $I \leq D^*$ and all $j = 1, ..., n$ takes $O(n(d^* D^* + D^* \log D^*))$. □

Again, replacing lost-sales with a backlogging linear cost function results in a mathematically identical situation where the inventory range is $[-D^*, D^*]$. Note that the running time of the model presented in this section, which is perhaps the most common model used throughout the literature, is computationally practical for a wide range of demands.

2.4 Approximation Schemes

In the previous section, we noticed that the main difficulty in the general stochastic lot sizing problem comes from the fact that the input can possibly contain very large numbers. For example, to compute the quantity $Y_j(z) = \sum_k P_{jk} \times r_j(z-k)$ for just one value of z in one period j would require considering all possible values for the demand k in that period, of which there can be arbitrarily many. The direct dependence of the running time on the quantities d^* and D^* is what gives rise to the hardness of the problem.

A natural way to think about approximating such a problem is to consider only a subset of the input values. If we can limit attention to a subset of the values that is bounded in size by a polynomial in the size of the input, we would be able to significantly enhance the performance of our algorithms. This speedup is gained at the cost of sacrificing some accuracy in the final solution, since we are discarding information by not looking at every value. We would like to bound the error resulting from using only a subset of the available information in such a way that we can still get a provably good solution in the end. Since we are considering fully polynomial time approximation schemes, 'provably good' here would require that we are at most a factor of ϵ away from the optimal solution, where ϵ is an accuracy parameter that the algorithm takes as input. Furthermore, the running time of our algorithms should depend polynomially on the size of the input and on $1/\epsilon$. The core of our approximation techniques is given in the following definitions and lemma.

Definition 13. *(δ-approximating set) Let $\delta > 0$, and let $f(.)$ be a non-negative, monotonically non-decreasing function defined on integers 0 to U. Suppose further*

that $f(0) = 0$ *and* $f(x)$ *is bounded below by* $\underline{U} \geq 1$ *for any* $x \neq 0$ *and above by* $f(U) = \overline{U}$. *A* δ*-approximating set of* $f(.)$ *is an ordered set* $S = \{i_1, ..., i_r\}$ *of integers fulfilling*

1. $S \subseteq \{0, 1, ..., U\}$;
2. *For each* $k = 1$ *to* $r - 1$, *if* $i_{k+1} > i_k + 1$, *then* $f(i_{k+1}) \leq (1 + \delta) f(i_k)$;
3. *Let* D *be the domain of* $f(.)$, *then for any element* $i \in D$, *there is an element* $j \in S$ *such that* $j \geq i$ *and* $f(j) \leq (1 + \delta) f(i)$.

Definition 14. *(*δ*-approximation) Consider a function* $f(.)$ *defined on integers* $[0, 1, ..., U]$. *Suppose this function fulfills the conditions given in Definition 13 and let* S *be a* δ*-approximating set for* $f(.)$. *A function* $\hat{f}(.)$ *defined as follows is called a* δ*-approximation of* $f(.)$. *For any integer* $x \in [0, U]$ *and* $\{i_k, i_{k+1}\} \in S$, *with* $i_k \leq x \leq i_{k+1}$

$$\hat{f}(x) = \begin{cases} f(i_{k+1}), & x \in [i_k, i_{k+1}], \text{ and } i_{k+1} \neq i_k + 1; \\ f(x), & x = i_k \text{ or } i_{k+1} \text{ and } i_{k+1} = i_k + 1. \end{cases}$$

Note that $\hat{f}(x)$ is well defined from Condition 3 in Definition 13, as there is guaranteed to be an interval in S into which any integer $x \in [0, U]$ falls. Let us put this definition in a more pictorial setting. Let D be the domain of $f(.)$, and let S be a subset of D fulfilling the points in Definition 13. We can think of $\hat{f}(.)$ as a step-approximation to $f(.)$: All integers in $[i_k, i_{k+1}]$ have the same value for $\hat{f}(.)$ for $i_{k+1} \neq i_k + 1$. Thus, $\hat{f}(.)$ generally looks like $f(.)$, but is more 'flattened'. We next show how to compute S.

Lemma 8. *(*δ*-approximation Lemma) Let* $f(.)$ *and* δ *be as defined above. A* δ*-approximating set of* $f(.)$ *can be computed in* $O(\frac{1}{\delta}\log(\max\{U, \overline{U}\}))$ *time.*

Proof. We give a procedure that computes the desired δ-approximating set in the time given in the lemma. Consider evaluating $f(.)$ at points 0 and U, if $f(U) \leq (1+\delta) f(0)$, we add $\{0, U\}$ to S and stop. We call $[0, U]$ a closed interval. Note that at this point,

$S = \{0, U\}$ fulfills all conditions in Definition 13. The first two conditions are easy to check, the third condition follows because $f(.)$ is non-decreasing, and hence if $f(U) \leq (1+\delta)f(0)$, then $f(U) \leq (1+\delta)f(x)$ as well, for any integer $x \in [0, U]$. If the condition $f(U) \leq (1+\delta)f(0)$ does not hold, then the interval $[0, U]$ is still 'open', and we split it into two new intervals $[0, \frac{U}{2}]$ and $[\frac{U}{2}+1, U]$. We recursively repeat the procedure for each of the new intervals, stopping whenever the condition is satisfied for an interval, closing that interval, and inserting its endpoints in their proper position in S. If the condition is not satisfied then the interval is open and we branch further. Since $f(.)$ is bounded above by $\overline{U}$, the maximum number of intervals that can be created before the condition is satisfied is $\log_{1+\delta}(\max\{U, \overline{U}\})$. This can be seen via a simple pigeonhole argument: One can divide the range $[0, \overline{U}]$ into at most $m = \lceil \log_{1+\delta}(\overline{U}) \rceil + 1$ disjoint boxes such that if $[a, b]$ are the end points of a box, then they satisfy $\frac{b}{a} = 1 + \delta$. Thus, the zeroth box contains just 0, the ith box has its end points $[(1+\delta)^{i-1}, (1+\delta)^i]$ for $i = 1, .., m-1$. Since the value for $f(.)$ for every element in its domain $[0, U]$ is in the range $[0, \overline{U}]$, then the functional value for any integer in $[0, U]$ will have to fall in one of the boxes. Consequently, any open interval $[x_1, x_2]$ will have the functional values corresponding to its end points lying in two different boxes, because if they lie in the same box then that would imply that $\frac{f(x_2)}{f(x_1)} \leq (1+\delta)$ and $[x_1, x_2]$ should be closed.

We consider two extreme cases that give rise to the terms in the max expression in the running time given in the statement of the lemma.

Case 1: Suppose U is less than $\overline{U}$. We can have at most as many open intervals at any one point as there are boxes. This is because an open interval has its two end points lying in different boxes (otherwise it would be closed), and all open intervals are disjoint. In the worst case, the binary branching will continue without any interval being closed until there are $\log_{1+\delta}(\overline{U})$ intervals. Since this will be a complete binary tree, the total number of intervals or nodes created in all tree levels above the last one will be at most $O(\log_{1+\delta}(\overline{U}))$ nodes, for an overall $O(\log_{1+\delta}(\overline{U}))$ number of nodes created. This in turn is equal to $\ln(\overline{U})/\ln(1+\delta)) \leq \lceil (1+1/\delta)\ln(\overline{U}) \rceil$ (from the Taylor series expansion of $\ln(1+\delta)$). The right hand side term in the last inequality is equal

to $\lceil (1+1/\delta)\log(\overline{U})/\log 2 \rceil = O(\frac{1}{\delta}\log(\overline{U}))$.

Case 2: Suppose U is greater than $\overline{U}$ and consider two adjacent values in the domain of the function that have a very large difference in their functional value that is not bounded by $(1+\delta)$. This indicates that both elements should be in S. In this case we will keep splitting the intervals so long as they contain those two values, until we reach those two adjacent values and include them in S as intervals of length one. To reach this point in our tree would take us $O(\log U)$ time. Thus the overall running time would be

$$O(\max\{\log U, \frac{1}{\delta}\log(\overline{U})\}) \leq O(\frac{1}{\delta}\log\max\{U, \bar{U}\})$$

□

Definition 13 and Lemma 8 are also true if $f(.)$ is monotone non-increasing instead of monotone non-decreasing. The second and third conditions in the definition are changed to

- For each $k = 1$ to $r-1$, if $i_{k+1} > i_k + 1$, then $f(i_k) \leq (1+\delta)f(i_{k+1})$;
- Let D be the domain of $f(.)$, then for any element $i \in D$, there is an element $j \in S$ such that $j \leq i$ and $f(j) \leq (1+\delta)f(i)$.

The branching procedure in the lemma is modified to account for this change in the obvious way: An interval $[a, b]$ is closed if $f(a) \leq (1+\delta)f(b)$, otherwise we continue branching on it. The following lemma is a simple, straightforward consequence of Definitions 13 and 14 that we will later use in deriving our FPTAS.

Lemma 9. *For any element x in the domain of $f(.)$, we can find an element $y \in S$ such that $f(y) \leq (1+\delta)f(x)$.*

Proof. The proof follows from the construction of S. Consider the case when $f(.)$ is monotone non-decreasing and recall that for any two consecutive elements i_k and i_{k+1}, both in S, we have $f(i_{k+1}) \leq (1+\delta)f(i_k)$. Therefore, for any x in the domain of f, we can find the interval $[i_k, i_{k+1}]$ into which it falls in S, and take y to be equal

to i_{k+1}. If $f(.)$ is monotone non-increasing then we choose y as i_k and the lemma still holds. □

Putting Definition 14 and Lemma 9 together, we arrive at the following simple corollary

Corollary 11. *Consider a function $f(.)$ and its δ-approximation $\hat{f}(.)$. For any x in the domain of $f(.)$, we have $\hat{f}(x) \leq (1+\delta)f(x)$.*

We now proceed to derive the FPTAS for the general single item stochastic lot sizing problem using the definitions and lemmas we discussed here. We will show how to approximate the various quantities introduced in the previous section, and how to put the various approximations together to get our desired result.

2.4.1 Approximating $Y_j(z)$

Recall from before that $Y_j(z)$ is the expected cost after a production decision has been made in period j, such that the production plus the incoming inventory is z. For fixed values of j and z, let us denote by $r_{jz}(k)$ the quantity $r_j(z-k)$. We can then write $Y_j(z)$ as

$$Y_j(z) = \sum_k p_{jk} \times r_{jz}(k)$$

To approximate $Y_j(z)$, we will approximate $r_{jz}(k)$ as well. The difficulty in calculating $r_{jz}(k)$ comes from having to consider all possible values for the demand k, so we will try to consider only a subset of these values. Observe that in addition to $g_{j+1}(.)$, the second component in the min expression for $r_{jz}(k)$ can be broken down into two components; the holding cost $h_j(z-k)^+$ and the lost sales cost $q_j(z-k)^-$. Notice that for fixed j and z and depending on the value of k, exactly one of those components will contribute a cost to $r_{jz}(k)$ while the other one will be 0. To approximate $r_{jz}(k)$ we will approximate both of these functions.

Let us first consider the case when the demand falls in the interval $[1, z]$. Here, $r_{jz}(k)$ will be equal to $h_j(z-k)^+ + g_{j+1}(z-k)$. Because we have assumed that we

can dispose of inventory for free, we would have $r_{jz}(k)$ be a monotone non-decreasing function in k. This is because the quantity $z - k$ decreases with increasing k and the values of $h_j(z-k)$ and $g_{j+1}(z-k)$ can only increase with decreasing $z-k$ (if $h_j(\alpha)$ was greater than $h_j(\beta)$ for $\alpha > \beta$, we can dispose of $\alpha - \beta$ items and have $h_j(\alpha) = h_j(\beta)$; the same is true for $g_{j+1}(.)$). We can thus use our δ-approximation Lemma to compute a δ-approximating set for $r_{jz}(k)$ in that range. We will denote the resulting set by $\Psi_j^1(z)$.

The case when the demand falls in the interval $[z, D_j]$ is completely analogous. In this interval, $r_{jz}(k)$ is equal to $q_j(k-z) + g_{j+1}(0)$, which is again a non-decreasing function in k. We compute a δ-approximating set for $r_{jz}(k)$ in this interval and denote it by $\Psi_j^2(z)$. A δ-approximating set for $r_{jz}(k)$ over all possible demand values is now given by appending $\Psi_j^2(z)$ to $\Psi_j^1(z)$. We denote the resulting δ-approximating set for $r_{jz}(k)$ by $\Psi_j(z)$.

For a fixed value of j and z, we Let $\overline{Y}_j(z)$ be an approximation to $Y_j(z)$. One can think of $\overline{Y}_j(z)$ as calculating $Y_j(z)$ over demand intervals instead of demand points. Every interval has a probability equal to the sum of the probabilities of all the demand points lying in it and each demand point is assigned a cost equal to the cost of the most expensive demand point in the interval. We can calculate $\overline{Y}_j(z)$ as follows

$$\overline{Y}_j(z) = \sum_{i_k \in \Psi_j(z)} (F_j(i_{k+1}) - F_j(i_k)) \cdot r_{jz}(i_{k+1}) \tag{2.3}$$

We can now apply the same techniques in order to compute a δ - approximating set for $\overline{Y}_j(z)$. We will denote the resulting set by Υ_j. We will make a slight abuse of notation and let $\hat{Y}_j(z)$ be a δ-approximation to $\overline{Y}_j(z)$, instead of the more correct $\hat{\overline{Y}}_j(z)$, to make it more easily readable.

To summarize; so far we have shown how to compute $\overline{Y}_j(z)$, an approximation to $Y_j(z)$ for fixed j and z. For a fixed value of j, We have also computed a δ-approximating set for the function $\overline{Y}_j(z)$ and denoted it by Υ_j. Finally, we denoted the δ-approximation for $\overline{Y}_j z$ by $\hat{Y}_j(z)$.

2.4.2 Approximating $g_j(I)$

We now proceed to approximate the function $g_j(I)$. First, consider the production cost function $c_j(.)$; we invoke Lemma 8 to compute a δ-approximating set for it and we denote the resulting set by Φ_j. Also, for a fixed value of I, we will denote $\hat{Y}_j(I+x)$ by $\hat{Y}_{jI}(x)$. We let $\overline{g}_j(I)$ be our approximation to $g_j(I)$, and we write $\overline{g}_j(I)$ as

$$\begin{aligned} \overline{g}_j(I) = \min \quad & c_j(x) + \hat{Y}_{jI}(y) \\ \text{st.} \quad & x \in \Phi_j, \quad y, y' \in \Upsilon_j(I) \\ & y \leq x \leq y' \end{aligned} \tag{2.4}$$

where y and y' are consecutive in $\Upsilon_j(I)$. One can see from the above that for fixed j and I, $\overline{g}_j(I)$ is simply $g_j(I)$ computed for only certain values of x; namely, those values that are in Φ_j. For every value $x \in \Phi_j$, there is a value $I + x$ that falls in some $[y, y']$ in $\Upsilon_j(I)$. Instead of calculating $\hat{Y}_{jI}(x)$, we calculate $\hat{Y}_{jI}(y) \geq \hat{Y}_{jI}(x)$ as $y \leq x \leq y'$ (recall that $\hat{y}_j(.)$ is non-increasing.)

We can now compute a δ-approximating set for $\overline{g}_j(I)$ by Lemma 8, we denote this set by $\Gamma_j(I)$. Again, we abuse the notation and let $\hat{g}_j(I)$ be our δ-approximation to $\overline{g}_j(I)$.

Let us now put these approximations together to derive a procedure that will lead to our FPTAS. Starting from period n and working backwards, we can compute the various δ-approximating sets at every stage. We now show that this sequence of approximations leads to an FPTAS. We start by proving a bound on the resulting approximation.

Lemma 10. *At stage j and for any value of I, we have $\hat{g}_j(I) \leq (1+\delta)^{3(n-j+1)} g_j(I)$.*

Proof. We prove the lemma by induction, starting with the base case for period n and working backwards. To make the following discussion easy to follow, we will break it down into a sequence of propositions.

Proposition 3. *For any z, if we calculate $\overline{Y}_n(z)$, then we have $\overline{Y}_n(z) \leq (1+\delta)Y_n(z)$.*

Proof. Consider the quantity $r_{nz}(k)$ and recall that we calculated it over demand intervals. Because $r_{nz}(k)$ is non-decreasing on k, any value for k that falls between i_k and i_{k+1} in $\Psi_n(k)$ has a value for $r_{nz}(.)$ that is at least equal to $r_{nz}(i_k)$. Thus a lower bound on the cost of $r_{nz}(.)$ for an entire demand interval $[i_k, i_{k+1}]$ can be given by $LB = (F_n(i_{k+1}) - F_n(i_k)) \cdot r_{nz}(i_k)$. From the construction of $\Psi_n(k)$, we know that i_k and i_{k+1} satisfy $r_{nz}(i_{k+1}) \leq (1+\delta) \cdot r_{nz}(i_k)$, and thus $(F_n(i_{k+1}) - F_n(i_k)) \cdot r_{nz}(i_{k+1}) \leq (1+\delta) \cdot LB$, the claim follows by summing the inequality over all demand intervals and noting that the resulting sum for LB is less than or equal to $Y_n(z)$. □

Proposition 4. *For any arbitrary z, we have $\hat{Y}_n(z) \leq (1+\delta)^2 Y_n(z)$.*

Proof. Because $\hat{Y}_n(z)$ is the δ-approximation to $\overline{Y}_n(z)$, this proposition is a direct consequence of Corollary 11 and Proposition 3.

□

Proposition 5. *For any I, we have $\overline{g}_n(I) \leq (1+\delta)^2 g_n(I)$.*

Proof. Let x^* be the minimizer for $g_n(I)$ (and hence z^* is equal to $I + x^*$). Note that $g_n(I)$ contains two terms $c_n(x)$ and $Y_n(z)$, and that by construction of Φ_n we can find an $x' \in \Phi_n$ such that $x' \geq x^*$ and $c_n(x') \leq (1+\delta)c_n(x^*)$. By proposition 4, we have $\hat{Y}_n(z^*) \leq (1+\delta)^2 Y_n(z^*)$. Since $z' = I + x' \geq I + x^* = z^*$ and $\hat{Y}_n(z)$ is non-increasing, we have $\hat{Y}_n(z') \leq \hat{Y}_n(z^*) \leq (1+\delta)^2 Y_n(z^*)$. Going back to $\overline{g}_n(I)$ in (2.4), we have $\overline{g}_n(I) = c_n(x') + \hat{Y}_n(z') \leq (1+\delta)c_n(x^*) + (1+\delta)^2 Y_n(z^*) \leq (1+\delta)^2 g_n(I)$, and the proposition is proved. □

Proposition 6. *For any arbitrary I, we have $\hat{g}_n(I) \leq (1+\delta)^3 g_n(I)$.*

Proof. Again, using Corollary 11, and Proposition 5, this result follows directly from the fact that $\hat{g}_n(I)$ is a δ-approximation of $\overline{g}_n(I)$. □

From the propositions above, the lemma holds true for the base case $j = n$. For the inductive step, we assume that the lemma is true for stage $j+1$, i.e. that for an arbitrary I, we have $\hat{g}_{j+1}(I) \leq (1+\delta)^{3(n-j)} g_{j+1}(I)$. Consider stage j and $g_j(I)$, we will

next derive the relationship between $g_j(I)$ and $\hat{g}_j(I)$. This derivation is essentially a repetition of the proofs we did for the base case, with the added precaution that we have the extra term $g_{j+1}(.)$ in the expression for $r_j z(k)$.

Let us pause for a moment to think about $r_{jz}(k)$, breaking it down into its constituent parts, we can repeat the analysis we did earlier for $r_{nz}(k)$. Recall that $r_{jz}(k) = \min\{r_j((z-k)-1), q_j((z-k)^-) + h_j((z-k)^+) + g_{j+1}((z-k)^+)\}$. However, because $g_{j+1}(.)$ is unavailable, we will use $\hat{g}_{j+1}(.)$ instead. Let us denote by $\hat{r}_{jz}(k)$ the function that is the same as $r_{jz}(k)$ but that uses $\hat{g}_{j+1}(.)$ instead of $g_{j+1}(.)$. From the induction hypothesis, if we evaluate both $r_{jz}(k)$ and $\hat{r}_{jz}(k)$ at the same value for k, we will have $\hat{r}_{jz}(k) \leq (1+\delta)^{3(n-j)} r_{jz}(k)$.

As before, we can compute a δ-approximating set for $\hat{r}_{jz}(k)$ by Lemma 8 and we denote it by Ψ_j.

Proposition 7. *For any z, we have $\hat{Y}_j(z) \leq (1+\delta)^{3(n-j)+2} Y_j(z)$.*

Proof. The proof is exactly the same as the proofs in Propositions 3 and 4: Let us denote by $\dot{Y}_j(z)$ the value we get when we replace $r_{jz}(k)$ by $\hat{r}_{jz}(k)$ in (2.3). Because we calculate $\overline{Y}_j(z)$ over Ψ_j, we will have $\overline{Y}_j(z) \leq (1+\delta)\dot{Y}_j(z)$. But $\dot{Y}_j(z)$ itself is off from $Y_j(z)$ by the factor $(1+\delta)^{3(n-j)}$ due to using $\hat{r}_{jz}(k)$ in (2.3). Thus the overall error is $\overline{Y}_j(z) \leq (1+\delta)^{3(n-j)+1} Y_j(z)$. Since $\hat{Y}_j(z)$ is the δ-approximation to $\overline{Y}_j(z)$, then by Corollary 11 we have $\hat{Y}_j(z) \leq (1+\delta)\overline{Y}_j(z) \leq (1+\delta)^{3(n-j)+2} Y_j(z)$.

□

Finally, by duplicating the proofs to Propositions 5 and 6 and noting that the relative error between $g_j(I)$ and $\overline{g}_j(I)$ is the same as the relative error between $\hat{Y}_j(z)$ and $Y_j(z)$, we have $\overline{g}_j(I) \leq (1+\delta)^{3(n-j)+2} g_j(I)$. Given that $\hat{g}_j(I)$ is the δ-approximation to $\overline{g}_j(I)$ and by Corollary 11, we have $\hat{g}_j(I) \leq (1+\delta)\overline{g}_j(I) \leq (1+\delta)^{3(n-j)+3} g_j(I)$, which completes the proof.

□

2.4.3 Approximation guarantee and running time analysis

We are now in a position to prove the main result of the chapter.

Theorem 11. *The outlined approximation procedure gives an FPTAS for the general stochastic lot sizing problem when we set* $\delta = \frac{\epsilon}{6n}$.

Proof. We first prove that the resulting approximation is indeed a $(1+\epsilon)$ approximation. For stage 1, and by Lemma 10, we know that we have $\hat{g}_1(0) \leq (1+\delta)^{3n} g_1(0)$.

Setting δ to the value in the theorem, we get $\hat{g}_1(0) \leq (1+\frac{\epsilon}{6n})^{3n} g_1(0)$. Using the inequality $(1+x/n)^n \leq 1+2x$ for $0 \leq x \leq 1$, we get $\hat{g}_1(0) \leq (1+2\cdot\epsilon/2)\cdot g_1(0)$, i.e. $\hat{g}_1(0) \leq (1+\epsilon)\cdot OPT$.

We use Lemma 8 to Show that the algorithm runs in time polynomial in the input size and $1/\epsilon$. Let B be an upper bound on the cost and recall we denoted the maximum demand value by D^*. In any period, we calculate the δ-approximating sets Ψ_j, Υ_j, Γ_j. The running time of any one of these operations is bounded above as given by the lemma to be $O(\frac{1}{\delta}\log(B+D^*))$. To compute Γ_j for a fixed j we need to scan through $O(\frac{1}{\delta}\log(B+D^*))$ values of I. For each of these values, we need to compute Φ_j and Υ_j, which again takes $O(\frac{1}{\delta}\log(B+D^*))$. Thus for one period we need time proportional to $O(\frac{1}{\delta}^2\log^2(B+D^*))$. Doing this for the n periods will lead to an overall running time proportional to $O(\frac{n}{\delta^2}\log^2(B+D^*)$. This is also the time needed to compute $\overline{g}_j$ for all $I \in \Gamma_j$ and for all j. Putting δ as $\frac{\epsilon}{6n}$ as in the statement of the lemma gives us a running time of $O(\frac{n^3}{\epsilon^2}\log^2(B+D^*))$, which is polynomial in $1/\epsilon$ as well as the size of the data in the problem. This finishes the proof of the theorem. □

We have thus shown that under our assumptions, one can develop an FPTAS for the stochastic lot sizing problem. This is indeed a very strong result given that until very recently, heuristics with no provable guarantees were the only available results for this extremely important problem. In the next section, we discuss the possible extensions that we will add to our work

2.5 Conclusions and Future Work

We have mentioned that the lot sizing problem is very rich with variations that stem from the main problem we investigated in our work. Some of these variations which we are currently considering are the following. Developing an FPTAS for the model we discussed in this chapter, but under capacity constraints. Another direction is to include correlated demands but in a Markovian sense, where the distribution of the demand in period j depends on the actual realized demand in period $j - 1$. If we assume we have constant lead times that are not equal to zero, then its conceivable that we can still get an FPTAS but with running times that are –while still polynomial– largely unattractive. It is not clear that incorporating stochastic lead times can still get us an FPTAS.

In summary, we have used some of the ideas we introduced in chapter 1, along with some other techniques to develop our FPTAS in this chapter. The difficulty of the problem came from the many quantities that we need to approximate and keep track of, while making sure that at any point our relative error is still within control to obtain the ϵ approximation. The technique introduced in the δ-approximation Lemma would probably prove very useful for a larger multitude of problems (this is the same technique we have used to be able to get an FPTAS in chapter 1 for problems where the cardinality of our decision sets were exponential.) It would be interesting to see other applications for this method.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[2] Ellis Horowitz and Sartaj Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *J. ACM*, 23(2):317–327, 1976.

[3] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, 1975.

[4] Sartaj K. Sahni. Algorithms for scheduling independent tasks. *J. ACM*, 23(1):116–127, 1976.

[5] Gerhard J. Woeginger. When does a dynamic programming formulation guarantee the existence of an fptas? In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 820–829, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[6] C. H. Papadimitriou and M. Yannakakis. On the approximability of trade-offs and optimal access of web sources. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 86, Washington, DC, USA, 2000. IEEE Computer Society.

[7] G.V. Gens and E.V. Levner. Fast approximation algorithms for job sequencing with deadlines. *Discrete Applied Mathematics*, 3:313–318.

[8] M.Y. Kovalyov and W. Kubiak. A fully polynomial time approximation scheme for the weighted earliness-tardiness problem. *Operations Research.*

[9] Ravindra K. Ahuja, Özlem Ergun, James B. Orlin, and Abraham P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1-3):75–102, 2002.

[10] A.H.G. Rinnooy Kan E.L. Lawler, J.K. Lenstra and D.B. Shmoys. Sequencing and scheduling: Algorithms and complexity. In A.H.G. Rinnooy Kan S.C. Graves and P.H. Zipkin, editors, *Logistics of Production and Inventory*, pages 445–522. Handbooks in Operations Research and Management Science 4, North-Holland, Amsterdam, 1993.

[11] J.K. Lenstra R.L. Graham, E.L. Lawler and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[12] A.H.G. Rinnooy Kan J.K. Lenstra and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[13] P. Brucker and M.Y. Kovalyov. Single machine batch scheduling to minimize the weighted number of late jobs. *ZOR - Mathematical Methods of Operations Research*, 43:1–8, 1996.

[14] Zhi-Long Chen. Parallel machine scheduling with time dependent processing times. *Discrete Appl. Math.*, 70(1):81–93, 1996.

[15] Daniel W. Engels, David R. Karger, Stavros G. Kolliopoulos, Sudipta Sengupta, R. N. Uma, and Joel Wein. Techniques for scheduling with rejection. In *ESA '98: Proceedings of the 6th Annual European Symposium on Algorithms*, pages 490–501, London, UK, 1998. Springer-Verlag.

[16] Sudipta Sengupta. Algorithms and approximation schemes for minimum lateness/tardiness scheduling with rejection. In *Algorithms and Data Structures, 8th International Workshop, WADS 2003, Ottawa, Ontario, Canada, July 30 - August 1, 2003, Proceedings*, pages 79–90. Springer, 2003.

[17] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *J. ACM*, 21(2):277–292, 1974.

[18] E.L. Lawler. Fast approximation schemes for knapsack problems. *Mathematics of Operations Research*, 4:339–356, 1979.

[19] Eric Angel, Evripidis Bampis, and Alexander Kononov. On the approximate tradeoff for bicriteria batching and parallel machine scheduling problems. *Theor. Comput. Sci.*, 306(1-3):319–338, 2003.

[20] N. Erkip, W. H. Hausman, and S. Nahmias. Optimal centralized ordering policies in multi-echelon inventory systems with correlated demands. *Manage. Sci.*, 36(3):381–392, 1990.

[21] Hau L. Lee, Kut C. So, and Christopher S. Tang. The value of information sharing in a two-level supply chain. *Manage. Sci.*, 46(5):626–643, 2000.

[22] T Iida and P Zipkin. Approximate solutions of a dynamic forecast-inventory model. Working Paper, 2003.

[23] P. Zipkin. *Foundations of inventory management.* McGraw-Hill, 2000.

[24] Lingxiu Dong and Hau L. Lee. Optimal policies and approximations for a serial multiechelon inventory system with time-correlated demand. *Oper. Res.*, 51(6):969–980, 2003.

[25] A Veinot. Optimal policy for a multi-product, dynamic, nonstationary inventory problem. *Manage. Sci.*, 12:206–222, 1965.

[26] E. Chan. Markov chain models for multi-echelon supply chains. PhD thesis, 1999. School of ORIE, Cornell University.

[27] Retsef Levi Martin Pál, Robin Roundy and David Shmoys. Approximation algorithms for stochastic inventory control models. *IPCO*, 2005.