LOT SIZING
AND
DELIVERY SCHEDULING

Suquan Ju

LYNGBY 2003
MASTER THESIS
NR. 69/2003

IMM

Printed by IMM, DTU

Preface

This M.Sc thesis has been prepared by student Suquan Ju during the period from the 1^{st} of June, 2003 to the 30^{th} of November, 2003. The work has been carried out at the section Operations Research, at the Department of Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark (DTU).

This thesis is the final requirement to obtain the degree: Master of Science in Engineering. Readers of this thesis are assumed to have some knowledge in the area of operations research.

My supervisor is Professor Jens Clausen. I hereby thank Jens's instruction during this project.

Suquan Ju
Lyngby, November 30th, 2003

Abstract

The economic lot and delivery scheduling problem (**ELDSP**) involves a supply chain consisting of a supplier and an assembly facility, where direct shipments are made from one to the other. In this thesis, the common cycle case (**CCC**) of the ELDSP problem is presented. The objective of the CCC case is to find the production sequence and cycle duration that minimize the average cost per unit time as the sum of transportation costs, inventory holding costs at both the supplier and the assembly facility, and setup costs at the supplier.

Two different algorithms for the CCC case are compared both in solution quality and solution time. Previous experiments are only focused on small problem sizes with up to 9. In the project, experiments are done using different problem sizes ranging from 2 to 100. Furthermore, the complexity of Hahm and Yano's heuristic algorithm is estimated through experiments.

Keywords: Economic Lot, Delivery Scheduling, Common Cycle Case, Lot Sizing, Just-In-Time, Inventory Management.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Effective inventory control is an essential factor in the success of almost every organization. To help firms effectively manage their inventory, many inventory models have been developed. Just-In-Time (**JIT**) manufacturing is a successful model of inventory management. Having been developed and refined in Japan for decades, the JIT system of manufacturing was the device that allowed Japanese manufacturing companies to compete and eventually defeat American manufacturers throughout the 1970's, and 80's.

At the beginning of the 1990's, Hahm and Yano (USA) developed a heuristic procedure in the common cycle case (**CCC**) of the economic lot and delivery scheduling problem (**ELDSP**) defined by them [6] to find a JIT schedule. The heuristic algorithm can not always guarantee to obtain globally optimal solutions, however Hahm and Yano could not generate problems that make their algorithm fail. Jensen presented a new polynomial time algorithm in [14]. This time the globally optimal solutions are guaranteed to find. But the experiments conducted by both Hahm & Yano and Jensen are focused on small problem sizes with up to 9.

In this project, through the study of both Hahm & Yano's heuristic algorithm and Jensen's polynomial time algorithm, we experiment problems with problem sizes ranging from 2 to 100 for both algorithms. The performance of the algorithms are compared, with respect both to quality of solution and solution time. The complexity of Hahm and Yano's heuristic algorithm is estimated.

Chapter 2

The EOQ and EPQ Models

Economic Order Quantity (**EOQ**) and Economic Production Quantity (**EPQ**) models have been used for many years to optimize batch quantities in transportation and production.

If a product is purchased, the critical question is: "How much should we order each time?", and the answer is given by EOQ model. If a product is manufactured, the question becomes: "How much should we produce each time?", and the answer is given by EPQ model.

In this chapter, the EOQ and EPQ models are described, and the difference between these two models is explained. Furthermore, the average inventory level derived in each model will be used in Chapter 4.

2.1 Economic Order Quantity Model

The objective of EOQ model is to choose the optimal order quantity that minimizes the total annual cost as the sum of ordering cost and holding cost. The model formula is derived under a set of assumptions as follows:

2.1.1 Model Assumptions

Assumptions of the EOQ model are:

1. Only one product is involved.
2. Demand is known and constant.
3. Lead time is known and constant.
4. There are no quantity discounts.
5. Each order is received in a single delivery.

2.1.2 Model Parameters

The following parameters will be used in the EOQ model:

Q: order quantity.
T: cycle time.
D: annual demand.
C_o: ordering cost per order.
C_h: annual holding cost per unit.
TC: total annual cost, objective function.
d: demand rate, $\frac{Q}{T}$.
$O(t)$: cumulative order quantity from time 0 to time t; $t \neq nT$, where n is a non-negative integer.
$D(t)$: cumulative demand quantity from time 0 to time t.

2.1.3 Average Inventory Level

Both in EOQ and EPQ models, the **Average Inventory Level** is defined by geometrically calculating the area below the inventory level curve in a cycle time divided by the cycle time.

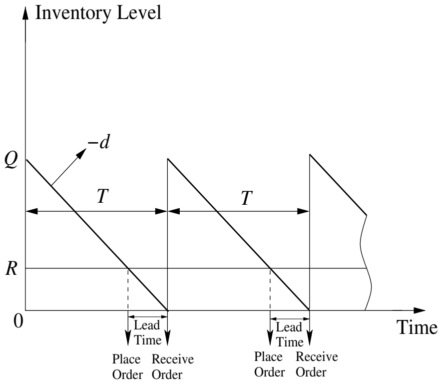

Figure 2.1: The inventory order cycle.

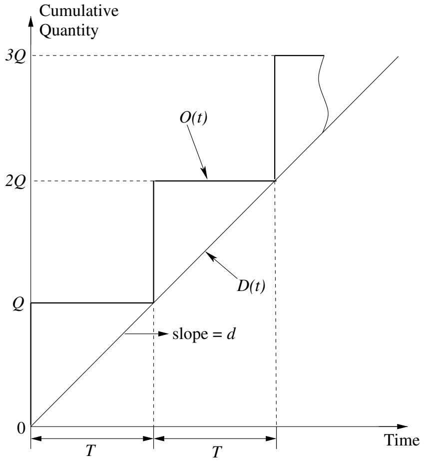

Figure 2.2: Cumulative quantity over time.

The continuous inventory order cycle system inherent in the EOQ model is reflected in Figure 2.1. An order quantity, Q, is received and is used up over time at a constant rate, d. When the inventory level decreases to the reorder point, R, a new order is placed; a period of time, referred to as the *lead time*, is required for delivery. The order is received all at once just at the moment when demand depletes the entire stock of inventory, so there will be no shortages. This cycle is repeated continuously for the same order quantity, reorder point, and lead time.

Figure 2.1 can be redrawn in another way in Figure 2.2, which shows the cumulative quantity of both order and demand over time.

According to the definition, the average inventory level in the EOQ model will be:

$$\begin{aligned} \text{Average inventory level} &= \frac{\text{area in a cycle time}}{\text{cycle time}} \\ &= \frac{\frac{1}{2}QT}{T} \\ &= \frac{Q}{2} \end{aligned} \tag{2.1}$$

2.1.4 Model Formulation

As we mentioned, the objective function of EOQ model is to minimize the total annual cost as the sum of ordering cost and holding cost. These two costs react inversely to each other. As the order quantity increases, fewer orders are required, which causes the annual ordering cost to decline. Whereas the average amount of inventory on hand will increase, resulting in an increasing in the holding cost. Thus, in effect, the optimal order quantity represents a compromise between these two costs.

Since the annual demand (D) is assumed to be known and constant, the number of orders per year will be D/Q. The total annual ordering cost is computed using the cost per order (C_o) times the number of orders per year, which is:

$$\text{Annual ordering cost} = \frac{C_o D}{Q}$$

The only variable in this equation is Q; both C_o and D are constant parameters. Thus, the relative magnitude of the ordering cost is dependent on the order size.

The total annual holding cost is computed by multiplying the annual holding cost per unit (C_h) times the average inventory level ($Q/2$), which is:

$$\text{Annual holding cost} = \frac{C_h Q}{2}$$

Then the total annual cost can be got:

$$\begin{aligned} TC &= \text{Annual ordering cost} + \text{Annual holding cost} \\ &= \frac{C_o D}{Q} + \frac{C_h Q}{2} \end{aligned}$$

The graph in Figure 2.3 shows the inverse relationship between ordering cost and holding cost, resulting in a convex total cost curve. In Figure 2.3, the optimal order quantity occurs at the point where the total cost curve is at a minimum, which coincides exactly with the point where the holding cost curve intersects the ordering cost curve. This enables us to determine

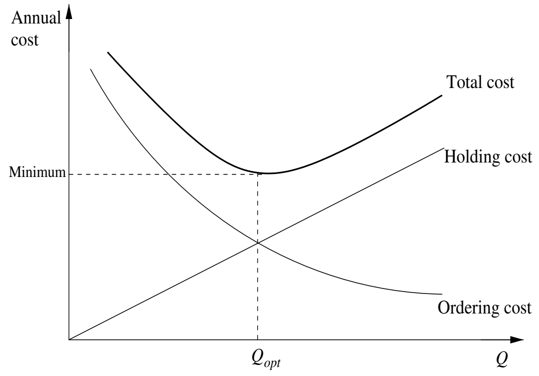

Figure 2.3: The EOQ cost model.

the optimal value of Q by equating the two cost functions and solving for Q:

$$\frac{C_o D}{Q} = \frac{C_h Q}{2}$$
$$\implies Q^2 = \frac{2C_o D}{C_h}$$
$$\implies Q_{opt} = \sqrt{\frac{2C_o D}{C_h}}$$

An alternative way of deriving Q_{opt} can be found in Appendix A.1.

Finally, the total minimum cost is determined by substituting the Q value for the optimal order size, Q_{opt}, in the objective function:

$$TC_{min} = \frac{C_o D}{Q_{opt}} + \frac{C_h Q_{opt}}{2}$$

2.2 Economic Production Quantity Model

The objective of the EPQ model is to choose the optimal production quantity that minimizes the total annual cost as the sum of setup cost and holding cost.

2.2.1 Model Assumptions

The assumptions of the EPQ model are similar to those of the EOQ model, except that instead of each order received in a single delivery, units are received incrementally during production.

2.2.2 Model Parameters

The following parameters will be used:

- Q: production quantity/lot size.
- T: cycle time.
- D: annual demand.
- T_p: production time.
- C_s: machine setup cost per setup.
- C_h: annual holding cost per unit.
- TC: total annual cost, objective function.
- d: demand rate ($\frac{Q}{T}$).
- p: production rate ($\frac{Q}{T_p}$).
- $P(t)$: cumulative production quantity from time 0 to time t.
- $D(t)$: cumulative demand quantity from time 0 to time t.

Here, Q refers to production quantity not order quantity, and C_o is substituted by C_s because we now don't purchase but we produce. Hence, there is no ordering cost but there is a setup cost because we must set up the machine before each production.

2.2.3 Average Inventory Level

Figure 2.4 reflects the EPQ model in a production inventory cycle system. During the production phase of a cycle time, inventory builds up at a rate

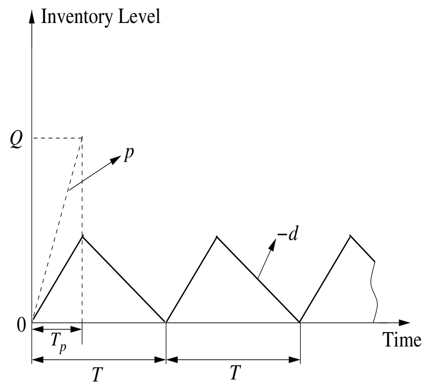

Figure 2.4: The inventory production cycle.

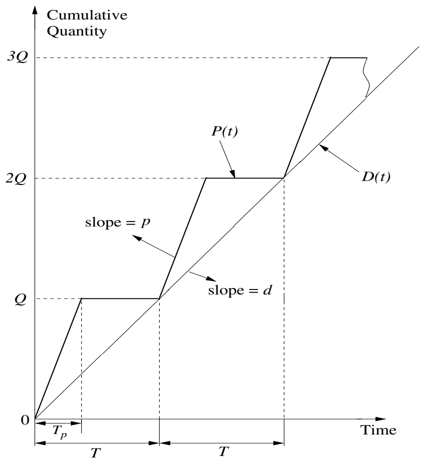

Figure 2.5: Cumulative quantity over time.

equals to the difference between production and demand rates $(p-d)$. As long as the production proceeds, the inventory will continue to build; when production ceases, the inventory level will begin to decrease at the rate d. Hence, the inventory level will be maximum at the point where production ceases. When the amount of inventory on hand is exhausted, production is resumed, and the cycle repeats itself. Figure 2.5 shows the cumulative quantity of production and demand over time.

The average inventory level in the EPQ model can be calculated:

$$\begin{aligned} \text{Average inventory level} &= \frac{\text{area in a cycle time}}{\text{cycle time}} \\ &= \frac{\frac{1}{2}T \cdot (p-d) \cdot \frac{Q}{p}}{T} \\ &= \frac{Q}{2}(1-\frac{d}{p}) \end{aligned} \tag{2.2}$$

2.2.4 Model Formulation

This time the objective function of EPQ model is changed to minimize the total annual cost as the sum of setup cost (instead of ordering cost) and holding cost. But the solution can be derived the same way as we did in EOQ model.

In case of purchasing (EOQ model) we dealt with the number of orders whereas in case of manufacturing we deal with the number of setups. The number of setups per year will be D/Q. The total annual setup cost is computed using the machine setup cost per setup (C_s) times the number of setups per year, which is:

$$\text{Annual setup cost} = \frac{C_s D}{Q}$$

The total annual holding cost is the same as the total annual holding cost in EOQ model except that the average inventory level in EPQ model becomes $\frac{Q}{2}(1-\frac{d}{p})$ not $\frac{Q}{2}$, and:

$$\text{Annual holding cost} = \frac{C_h Q}{2}(1-\frac{d}{p})$$

Then the total annual cost is:

$$\begin{aligned} TC &= \text{Annual setup cost} + \text{Annual holding cost} \\ &= \frac{C_s D}{Q} + \frac{C_h Q}{2}(1-\frac{d}{p}) \end{aligned}$$

Figure 2.3 is also available for the EPQ model, the only thing changed is that the setup cost curve should be used instead of the ordering cost curve.

Furthermore, the optimal production quantity can be got the same way as in the EOQ model (also can be done as the way in Appendix A.1):

$$\frac{C_s D}{Q} = \frac{C_h Q}{2}(1 - \frac{d}{p})$$
$$\implies Q^2 = \frac{2C_s D}{C_h(1 - \frac{d}{p})}$$
$$\implies Q_{opt} = \sqrt{\frac{2C_s D}{C_h(1 - \frac{d}{p})}}$$

Finally, the optimal value of objective function in the EPQ model is:

$$TC_{min} = \frac{C_s D}{Q_{opt}} + \frac{C_h Q_{opt}}{2}(1 - \frac{d}{p})$$

Chapter 3

Literature Review of the ELDSP Problem

The economic lot and delivery scheduling problem (**ELDSP**) was first defined by Hahm and Yano in [5] where they presented a very simplified version of the problem called the single item case. The objective of the ELDSP problem is to minimize the average cost per unit time as the sum of production setup costs at the supplier, inventory holding costs at both the supplier and the customer, and transportation costs.

In the succeeding work, Hahm and Yano extended this problem to the more general versions: the common cycle case [6], models for nested schedules [7], and powers of two policies [8]. Because the ELDSP problem presented by Hahm and Yano assumes that the production rate of each component is constant and all components are of acceptable quality, Khouja developed an algorithm in [11] using a volume flexible production system where component quality depends on both lot sizes and unit production times.

Likewise, many other cases can happen depending on different situations we meet. However, with the development of the world's industry, more and more complicated cases maybe occur in the future. In order not to lose the generalization, in Chapter 4 we will focus on the classical case: the common cycle case and compare two different types of algorithms for this case.

3.1 The Single Item Case

The single item case [5] concerns only a single component or product produced on a single machine at the supplier and delivered at regular intervals to the customer, which is typically assembly facility, at a fixed delivery cost per shipment. The problem is to determine the frequency of production of a single component and the frequency of delivery of that component to a customer that uses this component at a constant rate, which eventually minimize the objective function. Only a very simplified version of the problem called the single item case was specified, but there are many applications for which this model captures the essence and fundamental trade-offs.

For this simple model, a procedure which provides optimal solutions was developed. It's proved that the ratio between the production interval and delivery interval must be an integer in an optimal solution. The analysis also provided a basis for determining when it is optimal to synchronize production and delivery, and how much setup costs and setup times must be reduced in order for synchronization to be optimal.

In the follow up of the simplest model, the authors extended this case to more general versions: the common cycle case [6], models for nested schedules [7] and powers of two policies [8].

3.2 The Common Cycle Case

In the common cycle case [6], the supplier produces multiple components instead of only one component and then delivers them to the assembly facility. The supplier produces exactly one batch of each component between deliveries, where the batch sizes are exactly equal to the assembly facility's requirements until the subsequent delivery arrives. Because of the setup costs and setup times when changing the production it can never be advantageous to produce the same component twice during the same cycle.

The common cycle case is discussed in details in Chapter 4.

3.3 Models for Nested Schedules

The model in the common cycle case [6] is generalized in models for nested schedules [7] to consider multiple deliveries per cycle, while retaining the assumption of one production run of each component in each cycle time. This generalization creates the need to partition the components into groups and to sequence the groups of components.

Hahm and Yano developed a heuristic solution procedure to simultaneously find a delivery interval and a common production cycle (interval) for all components that is an integer multiple of the delivery interval. Thus, during the production cycle, each component is produced once, while one or more deliveries are scheduled during the cycle.

The authors also developed a lower bounding approach for this case. The experimental results indicate that the heuristic provides solution close to the lower bound in most instances. The results also show that a Just-In-Time schedule with equal production and delivery quantities may not necessarily be optimal. Thus, it may be desirable in many instances to allow production quantities to be greater than the delivery quantities until setup costs and times are reduced.

3.4 Powers of Two Policies

The last stage of Hahm and Yano's ELDSP problem is the powers of two policies [8], which is the most general case comparing to the aforementioned models. In this paper, the authors developed a heuristic procedure to find a cyclic production and delivery schedule with the power-of-two property. That is, in each cycle, each component is produced 2^{μ} times for some small integer μ, where the value of μ may differ across components. In addition, several equally-spaced deliveries occur in each cycle, where the number of deliveries is equal to the least common multiple of the component production frequencies.

Here we can see that the common cycle case [6] is a special case of this problem in which there is exactly one production run of each component and exactly one delivery per cycle.

Computational results suggest that the heuristic performs well in an absolute sense, and that significant savings can be achieved by using coordinated

production and delivery schedules rather than approaches in which they are decided sequentially. The results also indicate that in many situations, pure Just-In-Time policies are far from optimal. The model provides a basis for determining the type and extent of improvements needed in the quest for Just-In-Time.

3.5 Common Cycle, Rework, and Variable Production Rate

Previous solutions to the ELDSP problem assume a constant production rate for each component and that all components are of acceptable quality. These assumptions ignore volume flexibility and quality cost. Volume flexibility permits a system to adjust the production rate upwards or downwards within wide limits. Also, component quality may deteriorate with large lot sizes and decreased unit production times. In [11], Khouja developed an algorithm solving the ELDSP problem in the common cycle case [6] for a supplier using a volume flexible production system where component quality depends on both lot sizes and unit production times. The algorithm is tested using numerical examples.

The paper extends the ELDSP problem to the case of imperfect quality. Two cases of output quality are considered. In the first case, output quality deteriorates with large lot sizes but is independent of unit production times which are fixed. In the second case, output quality deteriorates with large lot sizes and decreased unit production times. For the first case, the explicit consideration of quality using rework cost results in smaller cycle time and lot sizes which support Just-In-Time inventory management. For the second case, an algorithm that was found to perform well for a broad range of problems was developed. Linking quality to unit production times resulted in changing the unit production times and sequence of production. Components with large rework cost will be produced at slower rates than others and, other things being equal, will be produced earlier in the sequence.

Chapter 4

Solving the ELDSP Problem in the Common Cycle Case

The common cycle case (**CCC**) is one of the versions in the ELDSP problem, which is introduced by Hahm and Yano [6] in 1995. In the paper, they analyzed the interface between a supplier and a customer, where direct shipments are made from one to the other. The final manufacturing step at the supplier involves multiple components produced on a single machine or production line. The components are accumulated and then delivered directly to the customer which uses these components at a constant rate. The supplier incurs a sequence-independent setup cost and/or setup time each time the production line is changed over from one component to another. On the other hand, setup costs and times for the customer are negligible.

Hahm and Yano developed a heuristic procedure to find a Just-In-Time (**JIT**) schedule in which one production run of each product and a subsequent delivery of these products to the customer occurs in each cycle. The objective is to find the cycle duration and the production sequence that minimize the average cost per unit time as the sum of transportation costs, inventory holding costs at both the supplier and the customer, and production setup costs at the supplier.

4.1 Introduction

JIT is a Japanese management philosophy which has been applied in practice since the early 1970's in many Japanese manufacturing organizations. It was first developed and perfected within the Toyota manufacturing plants by Taiichi Ohno as a means of meeting consumer demands with minimum delays. Having been developed and refined in Japan for decades, the JIT system of manufacturing was the device that allowed Japanese manufacturing companies to compete and eventually defeat American manufacturers throughout the 1970's, and 80's.

By the beginning of the 1990's, the American manufacturing industry was left wondering – What happened? As a result, recently JIT system of manufacturing is paid more and more attention and on the rise in American Industries. Meanwhile, the importance of coordinating the scheduling of production and outbound deliveries has been highlighted by the recent adoption of JIT systems in major American industry segments. A majority of suppliers feel that they are being forced to hold inventory costs for their customers, which are typically assembly facilities. These results suggest that suppliers are producing components in batches that are larger than the delivery quantities. On the other hand, a JIT schedule would have the suppliers producing only what is required for the next shipment. The characteristics such a schedule should have will be investigated later in this chapter.

The authors investigated the linkage between a large assembly facility such as an automotive assembly plant and its immediate suppliers, often referred to as 'first tier' suppliers. Because of the magnitude of their impact on total production and transportation costs, the authors focused on first tier suppliers that are large enough to justify direct shipments between the supplier and the assembly facility.

Here we consider a situation where the supplier is 'captive' and supplies multiple components to only one assembly plant. Such captive suppliers exist in many industries.

Figure 4.1 is an example of the inventory levels of three components.

The supplier produces exactly *one* batch of each component between delivery, where the batch sizes are exactly equal to the assembly facility's requirements until the subsequent delivery arrives. From an economic perspective, this restriction is most realistic when the unconstrained natural

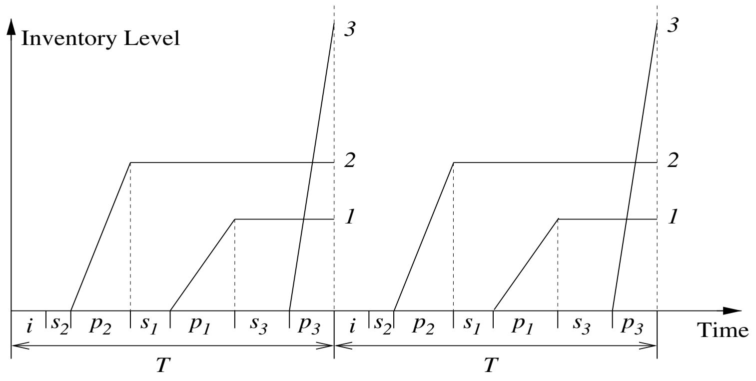

Figure 4.1: Inventory levels for two production cycles. The problem has three components labelled 1, 2 and 3. The actions of the machine are indicated on the time axis. i indicates idle time, s_x and p_x indicate setup time and production time of component x correspondingly.

setup intervals are similar in magnitude to the natural delivery intervals. However, the intent is to characterize good schedules with such a JIT requirement imposed exogenously.

Although a considerable amount of work has been done on multi-stage production systems with known constant demands, little research has been done that considers both the cost of inventory accumulation before delivery and the cost of transportation in the determination of jointly optimal production and delivery schedules. One reason why the former has been ignored is that many models assume instantaneous production. Other papers that do incorporate capacity constrains ignore some or all of the accumulation inventory, or treat transportation costs as fixed. But none explicitly considers the impact of the selected delivery schedule on the supplier's inventory levels.

The CCC case might be viewed as an extension of the 'product cycling' problem first posed by Hanssmann [10] in which several products, each with constant demand rates, are produced on one machine. Each product is produced once in each cycle and the cycle is repeated. Setup times and costs may be incurred in changeovers between products. The objective is to minimize average setup and inventory costs per unit time. Hanssmann shows that a closed form solution exists for this case.

The CCC case differs from Hanssmann's in two ways. First, demand occurs at a location other than where the components are produced. Thus, inventory of the components must be accumulated before each delivery. Second, because of this inventory accumulation between deliveries, the sequence in which components are produced affects the solution, even when setup times and costs are sequence-independent. One might argue that if production costs per unit time are similar across products, the production sequence may not matter in practice, because the rate at which value is being added is relatively constant. The key issue, however, is that there is inventory accumulation at the supplier between deliveries, and the cost of holding these inventories may be as substantial as the cost of holding inventories at the assembly facility. The magnitude of these inventories is influenced by the delivery frequency, and therefore should be included in this decision.

4.2 Problem Statement

It is useful to point out that in the CCC case, because of the setup costs and setup times while changing the production it can never be beneficial to produce the same component more than one time during the same cycle. Thus, under the JIT assumption, we don't need to make decisions about the number of production runs per delivery interval.

Because of the holding costs, it will always be optimal to start production at the latest possible time within each cycle. Thus, what is needed to solve is the cycle time and the production sequence within each cycle. Since the objective is to minimize the total cost, there is a tradeoff between the setup and transportation costs on one side and the holding costs on the other side. Minimizing setup and transportation costs requires production of large batches meaning long cycle times, while minimizing holding costs means keeping inventory levels low, i.e., short cycle times.

4.2.1 Assumptions

Initially, it is assumed that there is a fixed charge for each delivery. Deliveries to the assembly facility occur at equal intervals (duration to be determined). The delivery lead time is assumed to be deterministic, and without loss of generality, equal to zero.

Because the deliveries are equally spaced in time and the demand rates are constant, each shipment has the same composition. Inventory holding costs are charged on time-weighed average inventory levels, and the inventory holding cost of a component is assumed to be the same at the supplier as it is at the assembly facility.

The assumptions are listed in the following:

1. The supplier produces the components on a single machine or production line one at a time.
2. Production and demand rates are deterministic and constant.
3. The supplier incurs sequence independent setup costs and times.
4. There is a single delivery from the supplier to the assembly facility per cycle, during which one batch of each component is produced.
5. The setup costs and times for the assembly facility are negligible.
6. Both the supplier and the assembly facility incur inventory holding cost.
7. The delivery charge is fixed per shipment.
8. The delivery lead time is assumed to be deterministic equal to zero.

4.2.2 Notation

J: number of components.
A: transportation cost per delivery.
D_i: demand for component i per unit time.
S_i: setup cost for component i.
s_i: setup time for component i.
p_i: production time per unit of component i.
h_i: inventory holding cost per unit of component i per unit time.
T: cycle time, a decision variable.
q: production sequence vector. q contains a permutation of one of all possible production sequences, while $q[i]$ is the index of the component produced in the i^{th} position in the sequence. A decision variable.
TC the average cost per unit time as the sum of transportation costs, inventory holding costs at both the supplier and the customer, and production setup costs at the supplier, objective function.

We first develop the capacity constraint. During the cycle interval (T), components necessary for the next delivery are produced. The delivery quantity of component i is D_iT, so the total setup and production time for component i is $s_i + p_iD_iT$. The time needed to produce all components for one delivery is $\sum_{i=1}^{J}(s_i + p_iD_iT)$, which must be less than or equal to T. Therefore, we must have

$$\sum_{i=1}^{J}(s_i + p_iD_iT) \leq T,$$

which is equivalent to

$$T \geq \frac{\sum_{i=1}^{J} s_i}{1 - \sum_{i=1}^{J} p_iD_i} = \tau_{min} \tag{4.1}$$

Formula 4.1 gives a lower bound of a cycle time: τ_{min}.

4.2.3 Inventory Holding Costs

Inventory holding costs are incurred at both the supplier and the assembly facility. Because there is no production at the assembly facility, the average inventory holding cost of component i at the assembly facility is $\frac{1}{2}TD_ih_i$, just as it is in the EOQ model (equation 2.1). So the average inventory holding cost of all components per unit time at the assembly facility is:

$$\sum_{i=1}^{J} \frac{1}{2}TD_ih_i = \frac{1}{2}T\sum_{i=1}^{J} D_ih_i$$

On the other hand, the average inventory holding cost of component i at the supplier depends upon the accumulation during the production time of component i itself and the amount of time the inventory must wait during production of other components produced after i but before delivery. Then the average inventory cost per unit time at the supplier is as follows:

$$
\begin{aligned}
&\sum_{i=1}^{J} D_{q[i]}h_{q[i]}\left\{\frac{1}{2}TD_{q[i]}p_{q[i]} + \sum_{j=i+1}^{J}(TD_{q[j]}p_{q[j]} + s_{q[j]})\right\} \\
&= \sum_{i=1}^{J} D_{q[i]}h_{q[i]}\left(\frac{1}{2}TD_{q[i]}p_{q[i]}\right) + \\
&\sum_{i=1}^{J}\left\{D_{q[i]}h_{q[i]}\sum_{j=i+1}^{J}(TD_{q[j]}p_{q[j]} + s_{q[j]})\right\} \qquad (4.2)
\end{aligned}
$$

The first term in the formula (4.2) is the inventory holding cost per unit time incurred during production of $q[i]$, whereas the second term reflects the cost per unit time of holding $q[i]$ until delivery takes place. Because the first term in formula (4.2) is a sum of all components, this term is sequence independent and formula (4.2) can be further written as:

$$
\begin{aligned}
&\frac{1}{2}T\sum_{i=1}^{J} D_i^2 p_i h_i + \sum_{i=1}^{J}\left\{D_{q[i]}h_{q[i]}\sum_{j=i+1}^{J}(TD_{q[j]}p_{q[j]} + s_{q[j]})\right\} \\
&= \frac{1}{2}T\sum_{i=1}^{J} D_i^2 p_i h_i + \sum_{i=1}^{J} D_{q[i]}h_{q[i]}\sum_{j=i+1}^{J} s_{q[j]} + T\sum_{i=1}^{J} D_{q[i]}h_{q[i]}\sum_{j=i+1}^{J} D_{q[j]}p_{q[j]} \\
&= \frac{1}{2}T\sum_{i=1}^{J} D_i^2 p_i h_i + Z_1(q) + Z_2(q)\cdot T
\end{aligned}
$$

where

$$Z_1(q) = \sum_{i=1}^{J} D_{q[i]}h_{q[i]}\sum_{j=i+1}^{J} s_{q[j]}$$

and

$$Z_2(q) = \sum_{i=1}^{J} D_{q[i]}h_{q[i]}\sum_{j=i+1}^{J} D_{q[j]}p_{q[j]}$$

4.2.4 Formulation

As we mentioned, the objective function TC, which is the average cost per unit time as the sum of:

- **a**: production setup costs at the supplier;
- **b**: inventory holding costs at the supplier;
- **c**: inventory holding costs at the assembly facility; and
- **d**: transportation costs.

Write in formulation:

$$
\begin{aligned}
TC &= \underbrace{\frac{\sum_{i=1}^{J} S_i}{T}}_{\mathbf{a}} + \underbrace{\frac{1}{2}T\sum_{i=1}^{J} D_i^2 p_i h_i + Z_1(q) + Z_2(q)\cdot T}_{\mathbf{b}} + \underbrace{\frac{1}{2}T\sum_{i=1}^{J} D_i h_i}_{\mathbf{c}} + \underbrace{\frac{A}{T}}_{\mathbf{d}} \\
&= \frac{\sum_{i=1}^{J} S_i + A}{T} + \left(T\sum_{i=1}^{J} D_i^2 p_i h_i - \frac{1}{2}T\sum_{i=1}^{J} D_i^2 p_i h_i\right) + \\
&\quad Z_1(q) + Z_2(q)\cdot T + \frac{1}{2}T\sum_{i=1}^{J} D_i h_i \\
&= \frac{\sum_{i=1}^{J} S_i + A}{T} + \frac{1}{2}T\sum_{i=1}^{J} D_i h_i(1 - p_i D_i) + \\
&\quad Z_1(q) + Z_2(q)\cdot T + T\sum_{i=1}^{J} D_i^2 p_i h_i \\
&= \frac{S + A}{T} + \alpha T + Z_1(q) + Z_2(q)\cdot T + \beta T \qquad (4.3)
\end{aligned}
$$

where

$$S = \sum_{i=1}^{J} S_i$$

$$\alpha = \frac{1}{2}\sum_{i=1}^{J} D_i h_i (1 - p_i D_i)$$

$$\beta = \sum_{i=1}^{J} D_i^2 p_i h_i$$

subject to

$$T \geq \tau_{min}$$

The last inequality simply ensures that the cycle time is long enough to meet the demand.

Note that αT (see Appendix A.2) is the standard expression for the average inventory cost in the EPQ model (equation 2.2), and $Z_1(q) + Z_2(q) \cdot T + \beta T$ is the cost of inventory that is induced by the delivery schedule (because inventory must be accumulated at the supplier before each shipment). We can reexpress $Z_1(q) + Z_2(q) \cdot T + \beta T$ as shown below:

$$\begin{aligned}
&Z_1(q) + Z_2(q) \cdot T + \beta T \\
&= \sum_{i=1}^{J} D_{q[i]} h_{q[i]} \sum_{j=i+1}^{J} s_{q[j]} + T \sum_{i=1}^{J} D_{q[i]} h_{q[i]} \sum_{j=i+1}^{J} D_{q[j]} p_{q[j]} + T \sum_{i=1}^{J} D_i^2 p_i h_i \\
&= \sum_{i=1}^{J} D_{q[i]} h_{q[i]} \sum_{j=i+1}^{J} (T D_{q[j]} p_{q[j]} + s_{q[j]}) + \sum_{i=1}^{J} D_{q[i]} h_{q[i]} (T D_{q[i]} p_{q[i]}) \\
&= \sum_{i=1}^{J} D_{q[i]} h_{q[i]} \Big(T D_{q[i]} p_{q[i]} + \sum_{j=i+1}^{J} (T D_{q[j]} p_{q[j]} + s_{q[j]}) \Big) \\
&= \sum_{i=1}^{J} D_{q[i]} h_{q[i]} e_{q[i]}
\end{aligned}$$

Here $e_{q[i]} = T D_{q[i]} p_{q[i]} + \sum_{j=i+1}^{J} (T D_{q[j]} p_{q[j]} + s_{q[j]})$, which is the **Earliness** of the component produced in the i^{th} position in the sequence, and earliness is defined as the time between the start of production (after setup) of a component and the time of delivery.

4.2.5 Convexity

If we differentiate the objective function TC (equation 4.3) with respect to T, we can get:

$$\frac{\partial TC}{\partial T} = -\frac{S + A}{T^2} + \alpha + \beta + Z_2(q)$$

Since

$$\frac{\partial^2 TC}{\partial T^2} = \frac{S + A}{2T^3}$$

can be seen to be positive for $T > 0$, the minimum of TC with respect to T for a fixed production sequence q can be found at:

$$\frac{\partial TC}{\partial T} = 0 \Rightarrow T = \sqrt{\frac{S + A}{\alpha + \beta + Z_2(q)}} = T^*(q) \tag{4.4}$$

In the following, $T^*(q)$ will be used to denote the optimal cycle time for processing sequence q. As a result, if a sequence is given, the objective function is convex in $T^*(q)$.

From (4.4), we can also get a trivial upper bound on the optimal production cycle time T. Since $Z_2(q) > 0$, any optimal cycle time T^* will satisfy

$$T^* < \sqrt{\frac{S + A}{\alpha + \beta}} = \tau_{max}$$

4.3 Example

Having explained so far, we would like to show a small example of 3 components. The data selected in the objective function are listed in Table 4.1. Moreover, the data selected obey the rules in section 6.1.3.

	1	2	3
S_i	1.3	0.8	0.9
s_i	0.11	0.06	0.05
p_i	0.4	0.4	0.6
h_i	0.9	0.2	0.8
D_i	0.3	0.8	0.5
A	2		

Table 4.1: Example data of 3 components.

The results are showed in Table 4.2.

$\tau_{min} = 0.846154$	$\tau_{max} = 3.11045$	
q	$T^*(q)$	TC
1 2 3	2.61318	3.86445
1 3 2	2.48115	4.08408
2 1 3	2.74204	3.68602
2 3 1	**2.81272**	**3.62488**
3 1 2	2.53315	4.03186
3 2 1	2.64999	3.85919

Table 4.2: Results of 3 components.

From the result Table 4.2, we can see that the minimum of the objective function can be got at sequence $q = (2\ 3\ 1)$. We can enumerate all the possible sequences, which is $3! = 6$ sequences in this example, in order to get the optimal solution. But for large problem sizes, to calculate all the $J!$ sequences is a huge work and unwise. An algorithm is really needed to solve the CCC case in an efficient way, and the algorithms are described in Chapter 5.

Chapter 5

Solution Approaches

In this chapter, two algorithms that solve the ELDSP problem in the CCC case are introduced. They are Hahm & Yano's heuristic algorithm [6] and Jensen's polynomial time algorithm [14]. The reason that the heuristic algorithm fails in some situations is analyzed.

5.1 Hahm and Yano's Heuristic

As we mentioned in the beginning of Chapter 4, in order to minimize the objective function, we have to find both the cycle duration and the production sequence. We first consider the problem of finding the best production sequence for a given cycle time T. The only sequence-dependent parts of the objective function TC (equation 4.3) is $Z_1(q) + Z_2(q) \cdot T$, so in order to minimize TC, we must minimize this expression:

$$\begin{aligned} &Z_1(q) + Z_2(q) \cdot T \\ &\qquad = \sum_{i=1}^{J} D_{q[i]} h_{q[i]} \sum_{j=i+1}^{J} s_{q[j]} + T \sum_{i=1}^{J} D_{q[i]} h_{q[i]} \sum_{j=i+1}^{J} D_{q[j]} p_{q[j]} \\ &\qquad = \sum_{i=1}^{J} \left\{ D_{q[i]} h_{q[i]} \sum_{j=i+1}^{J} (T D_{q[j]} p_{q[j]} + s_{q[j]}) \right\} \end{aligned}$$

According to theorem 2.4 of [1], the value is minimized by arranging components in non-increasing order of $\frac{TD_{q[i]}p_{q[i]}+s_{q[i]}}{D_{q[i]}h_{q[i]}}$. So we can get a processing sequence satisfying:

$$\frac{TD_{q[1]}p_{q[1]} + s_{q[1]}}{D_{q[1]}h_{q[1]}} \geq \frac{TD_{q[2]}p_{q[2]} + s_{q[2]}}{D_{q[2]}h_{q[2]}} \geq \ldots \geq \frac{TD_{q[J]}p_{q[J]} + s_{q[J]}}{D_{q[J]}h_{q[J]}} \quad (5.1)$$

This result implies, roughly speaking, that components with long production time and low holding costs are produced early in the sequence, whereas components with short production time and high holding costs are produced late in the sequence. This delay the accumulation of 'expensive' inventory until close to the delivery time. Given a cycle time T, the optimal production sequence for cycle time T found in this way will be denoted $q^*(T)$ in the following.

Observe that if the setup time is negligible compared to the production time, the best sequence is generally determined by the value of $\frac{p_i}{h_i}$, which are independent of T.

From the objective function, we know if a sequence q is given, the objective function is convex in $T^*(q)$. Moreover, the best sequence can be found by the procedure in linear inequalities 5.1. This leads us to the following heuristic algorithm which iterates between selecting the sequence q and selecting T.

5.1.1 The Algorithm

The heuristic algorithm is listed in Table 5.1. The complexity estimation of the heuristic algorithm is given at the end of this chapter.

Step 1:	(Initialization) Set $n = 0$. Sequence the components in non-increasing order of $\frac{p_i}{h_i}$. Let this sequence be $q^{(0)}$. Calculate $Z_2(q^{(0)})$.
Step 2:	Set $n = n + 1$. Let $T^{(n)} = max\Big(\tau_{min}, T^*(q^{(n-1)})\Big)$.
Step 3:	Determine a new sequence by sorting all components in non-increasing order of $\frac{T^{(n)} D_i p_i + s_i}{D_i h_i}$. Let this sequence be $q^{(n)}$. If $q^{(n)}$ is same as $q^{(n-1)}$, *stop*. If $T^{(n)} = \tau_{min}$, stop. Otherwise, calculate the value of $Z_2(q^{(n)})$ for the new sequence; go to step 2.

Table 5.1: Hahm and Yano's heuristic algorithm

5.2 Weakness of the Heuristic Algorithm

As mentioned by Hahm and Yano in [6], their algorithm is able to find the optimal solutions of all 72 problems generated by them. But it can not always guarantee optimal solutions. Now we are going to specify how and why the heuristic fails in some situations.

5.2.1 The Global Optimum

Hahm and Yano's heuristic approaches the optimum by alternately calculating new values between T and q (Table 5.1). An example of this process is illustrated in Figure 5.1. The total cost curves of three sequences q_1, q_2 and q_3 have been indicated on the figure, as functions of the cycle time T. The solutions considered by the algorithm have been marked by '•', and the moves made visualized with dashed arrows. In the example of Figure 5.1 the algorithm manages to find the globally optimal solution.

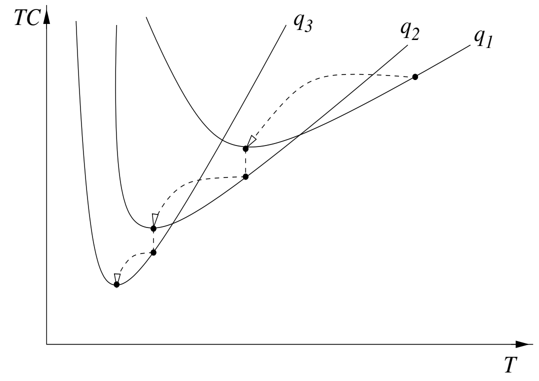

Figure 5.1: The alternative iterations are showed from Hahm and Yano's heuristic algorithm. In this example, the algorithm finds the global optimum.

5.2.2 The Local Optimum

Hahm and Yano's algorithm keeps running until it find a production sequence q' for which

$$q' = q^*(T^*(q')).$$

If a suboptimal solution satisfies this, the algorithm may get stuck in it. So the algorithm fails if a sequence $q^{(n)}$ is found which is suboptimal where the optimal cycle time $T^*(q^{(n)})$ got from the sequence $q^{(n)}$ does not lead to a new production sequence $q^{(n+1)}$ in the next step of the algorithm. This kind of local optimum is illustrated in Figure 5.2, along with the behavior of the algorithm.

We introduce the following notations:

$(T^*(q_g), q_g)$:	the global optimum.
$O = \{q \mid \exists T : q = q^*(T)\}$:	the set of production sequences optimal for some cycle times.
$C_1 = \{q \mid \exists T > T^*(q_g) : q = q^*(T)\}$:	the set of production sequences in O with optimal cycle times greater than the globally optimal cycle time.

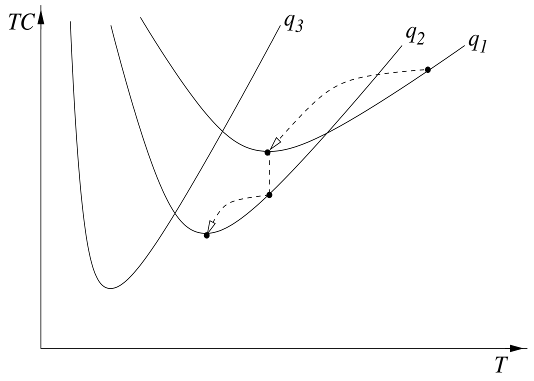

Figure 5.2: The alternative iterations are showed from Hahm and Yano's heuristic algorithm. In this example, the algorithm gets trapped in a local optimum.

5.2.3 Problem Analysis

In the following, two characteristics that make the heuristic fail will be discussed. Problems that make the heuristic fail but do not satisfy the following may exist.

The total cost curve of a production sequence $q \in C_1$ can basically be positioned in two different ways relative to the globally optimal production sequence. These have been illustrated on Figure 5.3.

In the left plot, the optimal choice of T for sequence q which is $T^*(q)$ locates under the total cost curve for the globally optimal sequence q_g. It will not be possible for Hahm and Yano's heuristic to move from q to q_g directly. Furthermore, if the algorithm cannot move to any other production sequence either, q may act as a local minimum, trapping the algorithm. Production sequences placed in this way satisfy:

$$\exists\, T_h : T^*(q_g) < T_h < T^*(q) \quad \text{and} \quad TC(T_h, q_g) = TC(T_h, q), \tag{5.2}$$

T_h is a local maximum located between the two optimal cycle times corresponding to their production sequences.

In the right plot of Figure 5.3 the total cost curve of q and q_g have been plotted in solid lines. The optimal choice of T for sequence q locates above

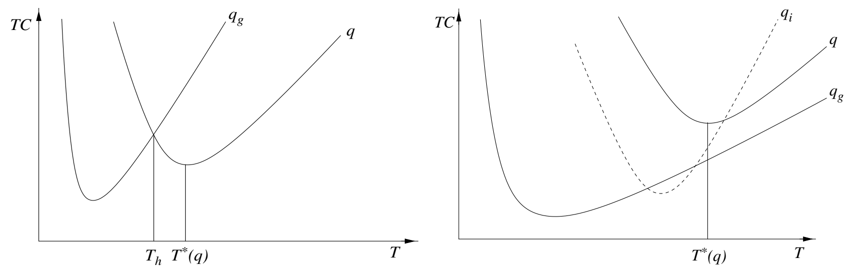

Figure 5.3: Illustration of the two ways of the total cost curve of a production sequence $q \in C_1$ can be placed relative to the total cost curve of the globally optimal q_g

the total cost curve for the globally optimal sequence q_g. Thus, it will be possible for the heuristic to move directly from q to q_g unless there is another production sequence q_i with a total cost curve located below the total cost curve of q at $T^*(q)$. Such a curve has been plotted dashed in the right plot. In order for a production sequence q placed in this way not to allow a direct move to the global optimum, the following must be satisfied:

$$\exists\, q_i \in C_1 : TC(T^*(q), q_g) < TC(T^*(q), q_i) < TC(T^*(q), q). \tag{5.3}$$

Since Hahm and Yano's algorithm can not move directly from any production sequence in C_1 satisfying (5.2) or (5.3) to the globally optimal production sequence, the algorithm is guaranteed to fail on the conditions for all $q \in C_1$, provided that the heuristic does not initialize $q^{(0)}$ to the globally optimal sequence.

5.3 Jensen's Algorithm

Having analyzed the weakness of Hahm and Yano's heuristic algorithm, Jensen presented a new polynomial time algorithm in his PhD dissertation [14]. This time the optimal solutions are guaranteed to be found for a given problem by Jensen's algorithm.

5.3.1 The Algorithm

Assume a production cycle time T is given, the corresponding production sequence q is easily found using formula (5.1). Suppose the cycle time is changed into $T' = T \pm \delta$, then the optimal production sequence q' corresponding to T' will be equal to q if formula (5.1) holds for T' and q. The continuous nature of formula (5.1) reveals that for a certain interval $[T_{min,i}, T_{max,i}]$ of T the optimal production sequence q_{opt} is fixed.

Due to the structure of formula (5.1), only a small number of intervals on T can exist. These interval intersections can be found by solving a number of equations of the form

$$\frac{TD_i p_i + s_i}{D_i h_i} = \frac{TD_j p_j + s_j}{D_j h_j} \tag{5.4}$$

Once the intersections have been found, the intervals on Ts can be found by sorting the intersections. Once the intervals have been constructed, the optimal production sequence q for each interval $[T_{min,i}, T_{max,i}]$ can be found using formula (5.1) with any value $t \in [T_{min,i}, T_{max,i}]$. When the optimal sequence has been found for the interval $[T_{min,i}, T_{max,i}]$, the optimal value of T within the interval will either be $T_{min,i}$, $T_{max,i}$ or be given by equation (4.4). Since the intervals are known to cover the entire search-space of T, we are certain that the global optimum will be found in this way. The algorithm is presented in a more structured way in Table 5.2.

Step 1: Calculate τ_{min} and τ_{max}.
Step 2: For each pair of components i, j solve equation (5.4) to find Ts. Store the T values larger than τ_{min} and smaller than τ_{max} in V.
Step 3: Insert τ_{min} and τ_{max} in V and sort them in increasing order, remove duplicates. Set $n = \text{size}(V)$.
Step 4: Set $TC_{best} = \infty$.
Step 5: for $(i = 1; i < n-1; i = i+1)$ $\{$

- ▷ set $T = \frac{1}{2}(V_i + V_{i+1})$.
- ▷ Calculate production sequence q corresponding to T, using equation (5.1).
- ▷ Calculate optimal cycle time $T^*(q)$ corresponding to q, using equation (4.4).
- ▷ Calculate total costs $TC_{T^*} = TC(q, T^*)$, $TC_{V_i} = TC(q, V_i)$ and $TC_{V_{i+1}} = TC(q, V_{i+1})$ using equation (4.3).
- ▷ if $TC_{T^*} < TC_{best} \wedge V_i < T^* < V_{i+1}$ then
  set $TC_{best} = TC_{T^*}, T_{best} = T^*, q_{best} = q$.
- ▷ if $TC_{V_i} < TC_{best}$ then
  set $TC_{best} = TC_{v_i}, T_{best} = V_i, q_{best} = q$.
- ▷ if $TC_{V_{i+1}} < TC_{best}$ then
  set $TC_{best} = TC_{V_{i+1}}, T_{best} = V_{i+1}, q_{best} = q$.

$\}$
Step 6: return TC_{best}, T_{best} and q_{best}.

Table 5.2: Jensen's polynomial time algorithm.

5.3.2 Complexity

Since at most $\frac{1}{2}J(J-1)$ different values of T can result from the solution of equation (5.4), at most $\frac{1}{2}J(J-1)+1$ different intervals can exist. Furthermore, these interval intersections can easily be found in time $O(J^2)$.

The time used in steps 4 and 6 is $O(1)$. The time used in step 1 is $O(J)$. The time used in step 3 is $O(J^2 \log J)$. The time used in step 2 is $O(J^2)$. The loop in step 5 runs $O(J^2)$ iterations, each iteration takes $O(J\log J)$ time. So the total time used in step 5 is $O(J^3 \log J)$. The time usage in step 5 is therefore dominant, and the overall time used by the algorithm is $O(J^3 \log J)$.

Now we try to estimate the complexity of Hahm and Yano's heuristic algorithm (Table 5.1).

The time used in step 1, step 2 or step 3 is the same, which is $O(J\log J)$. So the time used in the heuristic for each iteration is $O(J\log J)$. The number of iterations is unknown, but the potential number $\frac{1}{2}J(J-1)$ can be used, which is the maximum number of T values. The worst case running time used in the heuristic is $O(J^3 \log J)$, which is the same as Jensen's algorithm.

Chapter 6

Implementation

In order to demonstrate the difference between the solution approaches, both Hahm and Yano's heuristic [6] and Jensen's algorithm [14] have been implemented in C++. The experiments did by Hahm & Yano and Jensen are only for small number of components with up to 9. Dose the algorithms work for big number of components? In order to see the performance of the algorithms for big number of components, experiments are done from 2 components up to 100 components.

6.1 Random Problem Generation

In order to evaluate the impacts of various problem data on the performance of both algorithms, problem data are generated according to some rules.

6.1.1 Jensen's Generator

Since Hahm and Yano [6] could not generate problem data for which their heuristic algorithm failed to obtain a globally optimal solution, we first choose to use the same construction as Jensen did in his problem generator [14].

From the objective function (4.3), we can see that there are 6 types of parameters. In order to calculate the optimal solution of TC corresponding

to variable T and sequence q, the values of parameter A, S_i, s_i, h_i, p_i and D_i should be generated in advance.

In order to generate problems for which the heuristic fails we will focus on problems for which all the production sequences in C_1 are likely to satisfy (5.2). Requiring (5.2) to hold for all $q \in C_1$, solving for T_h and using equation (4.4) leads to:

$$\forall\ q \in C_1 : \sqrt{\frac{S+A}{\alpha+\beta+Z_2(q_g)}} < \frac{Z_1(q_g)-Z_1(q)}{Z_2(q)-Z_2(q_g)} < \sqrt{\frac{S+A}{\alpha+\beta+Z_2(q)}} \qquad (6.1)$$

The derivation of formula (6.1) is specified in Appendix A.3.

Formula (6.1) requires $Z_1(q_g) < Z_1(q)$ and $Z_2(q_g) > Z_2(q)$ for all $q \in C_1$, which are the same as:

$$\forall\ q \in C_1 : \sum_{i=1}^{J} D_{q_g[i]} h_{q_g[i]} \sum_{j=i+1}^{J} s_{q_g[j]} < \sum_{i=1}^{J} D_{q[i]} h_{q[i]} \sum_{j=i+1}^{J} s_{q[j]} \qquad (6.2)$$

and

$$\forall\ q \in C_1 : \sum_{i=1}^{J} D_{q_g[i]} h_{q_g[i]} \sum_{j=i+1}^{J} D_{q_g[j]} p_{q_g[j]} > \sum_{i=1}^{J} D_{q[i]} h_{q[i]} \sum_{j=i+1}^{J} D_{q[j]} p_{q[j]} \qquad (6.3)$$

Inequalities (6.2) and (6.3) are likely to hold if there is a component r for which s_r is much larger than s_i and p_r is much smaller than p_i for all $i \in \{1,\dots,J\}\backslash\{r\}$. In inequality (6.2) component r is scheduled at the very beginning in the global sequence q_g, so s_r is not included in the left part of inequality (6.2), which leads $\sum_{i=1}^{J} D_{q_g[i]} h_{q_g[i]} \sum_{j=i+1}^{J} s_{q_g[j]}$ is quite small. While in the local sequence q component r is scheduled not at the beginning of the sequence q, so s_r is included in the right part of inequality (6.2), which leads $\sum_{i=1}^{J} D_{q[i]} h_{q[i]} \sum_{j=i+1}^{J} s_{q[j]}$ is quite big. The similar principle can be used for p_r in inequality (6.3), component r should also be scheduled at the beginning in a global sequence q_g.

The generation rules are listed in Table 6.1. If not specified, in the following the value of F used during generating is equal to 1, the initial sequence of J components is $q = (1, 2, \dots, J)$.

F	a parameter used to increase the probability of generating problems satisfying (6.2) and (6.3), range: [1, ..., 100].
A	generated from a **Uniform Distribution** in $U(0,5)$;
S_i	drawn from a uniform distribution in $U(0,1)$;
h_i	drawn from a uniform distribution in $U(0,1)$;
s_i	drawn from a uniform distribution in $U(0,0.25)$. For one component r, s_r is generated using $s_r = F\sum_{i\neq r} s_i$;
p_i, D_i	generated from uniform distributions with values satisfying $0 < p_i \times D_i < \frac{1}{J}$ in order to insure feasibility. For one component r, p_r is generated using $p_r = \frac{1}{F} min_{i\neq r} p_i$.

Table 6.1: Jensen's generation rules.

6.1.2 Feasible Problems

One thing we have to mention here is that the problems generated should be feasible problems, because only feasible problems can lead to feasible solutions that we want to get eventually. In another word, since any optimal cycle time $T^* < \tau_{max}$ and any cycle time must have $T > \tau_{min}$, then any feasible problem should satisfy the condition: $\tau_{min} < \tau_{max}$. The derivations of τ_{min} and τ_{max} are got in Chapter 4.

In order to test the correctness of feasible problems in Jensen's generation rules, 100,000 problems are generated with the number of components ranging from 2 to 100. The feasibility condition is checked for all problems generated. Please see Table 6.2.

From Table 6.2 we observe that with the number of components increasing, it becomes more and more difficult for Jensen's generator to generate feasible problems which satisfy $\tau_{min} < \tau_{max}$. When the number of components are increased to 20, the probability of generating feasible problems is less than 0.1%. For the number of components 30, 50 and 100, Jensen's generator can not generate feasible problems. For 2, 3, 5 and even 7 components, the number of feasible problems is close to the number of problems generated. Still, infeasible problems occur strikingly often, especially for large components. Since only feasible problems are interesting, and Jensen did not check the feasibility condition in his experiments, the experimental results done by Jensen are data distortion.

J	No. of problems generated using Jensen's generator	No. of problems satisfying $\tau_{min} < \tau_{max}$
2	100000	98722
3	100000	98084
5	100000	91909
7	100000	73382
10	100000	30809
20	100000	33
30	100000	0
50	100000	0
100	100000	0

Table 6.2: Random problems generated using Jensen's generator.

6.1.3 A New Generator

We therefore must use some other ways to generate problems in order to get and compare feasible solutions.

If we consider Table 6.1 again, we can find the reason why Jensen's generator becomes more difficult to generate feasible problems while the number of components increasing. Because the drawing ranges never change when the number of components increase which make the difference between τ_{min} and τ_{max} become smaller and smaller, which again leads to the difficulty of generating feasible problems for big components. So to change the drawing ranges while the number of components increasing will be one way of removing that problem.

We know from Chapter 4 that $\tau_{min} = \frac{\sum_{i=1}^{J} s_i}{1-\sum_{i=1}^{J} p_i D_i}$ and $\tau_{max} = \sqrt{\frac{\sum_{i=1}^{J} S_i + A}{\alpha+\beta}}$. If we decrease s_i and increase S_i while drawing for the same number of components, τ_{min} and τ_{max} will be decreased and increased correspondingly. So the difference between τ_{min} and τ_{max} will be enlarged, which means the probability of generating feasible problems ($\tau_{min} < \tau_{max}$) is enhanced.

In the real world industry, it is natural that we should pay more for the setup costs if we want the setup time to be decreased. On the other hand, the amount we paid while decreasing the setup time should not be too expensive, which is not acceptable. Here we use $\sqrt{J}$ as the quotient of increase. For a 9 components example if the total setup time is decreased $\sqrt{J} = \sqrt{9} = 3$ times, we need to pay 2 times amount more for the total setup cost than the total setup cost for the normal setup time of 9 components. This is reasonable and acceptable. So every time we draw s_i for J components, the original drawing range of s_i is multiplied by $\sqrt{J}$. Similarly, S_i is got divided by $\sqrt{J}$. As a result, the total setup time is decreased $\sqrt{J}$ times, and the total setup cost is increased $\sqrt{J}$ times. The quotient of increase $\sqrt{J}$ is not unique, and many other quotients of increase can exist by selection.

In order to be clear at a glance, we write these two different rules in Table 6.3 to distinguish Jensen's generation rules.

S_i	drawn from a uniform distribution in $U(0, \sqrt{J})$;
s_i	drawn from a uniform distribution in $U(0, \frac{0.25}{\sqrt{J}})$. For one component r, s_r is generated using $s_r = F\sum_{i \neq r} s_i$;

Table 6.3: The new generation rules different from Jensen's.

Now as we can see from Table 6.3, the drawing ranges are changed by different number of components. In order to compare Jensen's generator, 100,000 problems are generated using the new generation rules. The experimental results are showed in Table 6.4.

J	No. of problems generated using the new generator	No. of problems satisfying $\tau_{min} < \tau_{max}$
2	100000	99454
3	100000	99737
5	100000	99933
7	100000	99976
10	100000	99997
20	100000	100000
30	100000	100000
50	100000	100000
100	100000	100000

Table 6.4: Random problems generated using the new rules.

From Table 6.4, we find that the number of feasible problems increase

while the number of components increasing, contrary to Jensen's generator. Furthermore, Jensen's data distortion phenomenon is well solved, especially for large components.

For the number of components from 2 to 10 there are still some problems that are not feasible. The reason can be explained that our quotient of increase $\sqrt{J}$ is not big enough, so sometimes the difference between τ_{max} and τ_{min} becomes negative. The phenomenon can be solved by increasing the quotient of increase, e.g. to J or even bigger.

From now on, we can generate large feasible problems using the new rules in order to compare the behaviors between Hahm & Yano's heuristic and Jensen's algorithm.

6.2 Two Examples of Seven Components

In order to illustrate the difference performance between Hahm and Yano's heuristic and Jensen's polynomial time algorithm, two examples of seven components will be illustrated together with the concrete parameters, which fulfill the new generation rules. Example A shows the same results for both algorithms, while Example B shows different results, i.e. Hahm and Yano's heuristic algorithm fails in Example B. The **boldfaced** values in the parameter table of each example are the values generated for component r.

6.2.1 Example A

The following parameters in Table 6.5 are used to implement both Hahm & Yano's heuristic and Jensen's algorithm in example A.

i	S_i	s_i	p_i	h_i	D_i
1	2.58261	0.039732	0.319407	0.537919	0.409467
2	1.58671	0.0494646	0.901578	0.252998	0.0690634
3	1.54214	0.0126077	0.733543	0.997772	0.0994598
4	2.51462	0.067139	0.00274667	0.144292	0.945647
5	2.43428	0.0439019	0.387494	0.784539	0.135258
6	1.939	**0.298786**	**0.00274667**	0.047853	0.634449
7	1.76774	0.0859409	0.181097	0.973907	0.304392
$A =$ 0.393384					

Table 6.5: Parameter values of Example A.

Table 6.6 is the procedures of implementing both algorithms, and the final results are listed in Table 6.7. Apparently these two algorithms obtain the exactly same results. For Hahm & Yano's heuristic algorithm (Table 5.1), $q^{(0)} = (2\ 3\ 1\ 5\ 7\ 6\ 4)$ is obtained by sorting $\frac{p_i}{h_i}$ in non-increasing order in **Step 1**. Then the program found a new sequence $q^{(1)} = (2\ 6\ 3\ 1\ 5\ 7\ 4)$ by sorting $\frac{T^{(1)} D_i p_i + s_i}{D_i h_i}$ in non-increasing order in **Step 3**, which is different from $q^{(0)}$. So **Step 2** and **Step 3** are repeated one more time until the end of **Step 3** when the stopping criterion is examined and shows that $q^{(2)}$ is the same as $q^{(1)}$, and the algorithm is stopped after two iterations.

Jensen's algorithm (Table 5.2) finds 5 intervals, so every optimal sequence in each interval is calculated. Then the algorithm selects the corresponding optimal TC in each interval as the optimal solution (from **A1** to **A6**). The final optimal solution is found in **A6**. In this example, we get the same solution for both algorithms.

$\boxed{\tau_{min} = 0.960552 \quad \tau_{max} = 5.52871}$

Hahm & Yano's algorithm

q	T
2 3 1 5 7 6 4	5.20509
2 6 3 1 5 7 4	5.16639

Jensen's algorithm

q	T	TC	
6 2 5 3 1 4 7	$T^* = 5.10452$	$TC_{T^*} = 5.8831$	
	$V_1 = 0.960552$	$TC_{V_1} = 16.0106$	← **A1**
	$V_2 = 1.1882$	$TC_{V_2} = 13.1955$	↩ **A2**
6 2 3 5 1 4 7	$T^* = 5.11601$	$TC_{T^*} = 5.87314$	
	$V_2 = 1.1882$	$TC_{V_2} = 13.1955$	
	$V_3 = 1.21106$	$TC_{V_3} = 12.9738$	↩ **A3**
6 2 3 5 1 7 4	$T^* = 5.14691$	$TC_{T^*} = 5.84667$	
	$V_3 = 1.21106$	$TC_{V_3} = 12.9738$	
	$V_4 = 1.99944$	$TC_{V_4} = 8.60737$	↩ **A4**
2 6 3 5 1 7 4	$T^* = 5.15552$	$TC_{T^*} = 5.84081$	
	$V_4 = 1.99944$	$TC_{V_4} = 8.60737$	
	$V_5 = 2.33637$	$TC_{V_5} = 7.72989$	↩ **A5**
2 6 3 1 5 7 4	$T^* = 5.16639$	$TC_{T^*} = 5.83421$	↩ **A6**
	$V_5 = 2.33637$	$TC_{V_5} = 7.72989$	
	$V_6 = 5.52871$	$TC_{V_6} = 5.84735$	

Table 6.6: Procedure of Example A.

Results	Hahm & Yano	Jensen
TC	5.83421	5.83421
T	5.16639	5.16639
q	2 6 3 1 5 7 4	2 6 3 1 5 7 4

Table 6.7: Final results of Example A.

6.2.2 Example B

The parameters used in Example B are listed in Table 6.8.

$i\diagdown$	S_i	s_i	p_i	h_i	D_i
1	0.613254	**0.448472**	**0.214576**	0.812769	0.0736412
2	2.113	0.0804214	0.915616	0.611988	0.122776
3	1.19268	0.0942835	0.340648	0.387921	0.140782
4	2.5726	0.0864051	0.334269	0.474105	0.233283
5	2.0149	0.0580148	0.301096	0.185949	0.358715
6	1.12711	0.0891533	0.541002	0.436048	0.240883
7	1.08827	0.0401934	0.214576	0.905057	0.41847
$A =$ 3.4077					

Table 6.8: Parameter values of Example B.

From Table 6.9 and Table 6.10 we can see that this time the results got from the two algorithms are different. 4 intervals are found by Jensen's algorithm. Hahm & Yano's algorithm behaves as in Example A, but this time the heuristic fails to obtain the optimal solution while it gets trapped into a local optimum.

From Table 6.8 we see that $r = 1$. Component r should be put at the 1^{st} position in a global sequence q_g, but the sequence $q = (5\ \mathbf{1}\ 2\ 6\ 3\ 4\ 7)$ found by Hahm and Yano's algorithm puts r in the second position of this sequence. The inequalities (6.2) and (6.3) are fulfilled, as a result the heuristic gets trapped in a local optimum q. The optimal sequence $q = (\mathbf{1}\ 5\ 2\ 6\ 3\ 4\ 7)$ found by Jensen's algorithm puts r at the first position of the sequence, then Jensen's algorithm finds the global optimum. However, the relative deviation, which is $\frac{5.91146-5.91143}{5.91143} \approx 0.0005\%$, is extremely small.

$\boxed{\tau_{min} = 2.1472 \qquad \tau_{max} = 5.52359}$

Hahm & Yano's algorithm

q	T
5 2 6 3 4 1 7	4.96433
5 1 2 6 3 4 7	4.89395

Jensen's algorithm

q	T	TC	
1 5 2 3 6 4 7	$T^* = 4.86318$	$TC_{T^*} = 5.91652$	
	$V_1 = 2.1472$	$TC_{V_1} = 7.96894$	$\leftarrow$ **B1**
	$V_2 = 2.42068$	$TC_{V_2} = 7.3889$	$\hookleftarrow$ **B2**
1 5 2 6 3 4 7	$T^* = 4.87167$	$TC_{T^*} = 5.91143$	$\hookleftarrow$ **B3**
	$V_2 = 2.42068$	$TC_{V_2} = 7.3889$	
	$V_3 = 4.88706$	$TC_{V_3} = 5.91146$	
5 1 2 6 3 4 7	$T^* = 4.89395$	$TC_{T^*} = 5.91146$	
	$V_3 = 4.88706$	$TC_{V_3} = 5.91146$	
	$V_4 = 5.21255$	$TC_{V_4} = 5.92294$	
5 2 1 6 3 4 7	$T^* = 4.9171$	$TC_{T^*} = 5.91316$	
	$V_4 = 5.21255$	$TC_{V_4} = 5.92294$	
	$V_5 = 5.52359$	$TC_{V_5} = 5.95207$	

Table 6.9: Procedure of Example B.

Results	Hahm & Yano	Jensen
TC	5.91146	5.91143
T	4.89395	4.87167
q	5 1 2 6 3 4 7	1 5 2 6 3 4 7

Table 6.10: Final results of Example B.

6.3 Running Time Comparison

In this section, we compare the running time of the two algorithms. Consider the Table 6.11, 100,000 feasible problems are generated using the new generation rules for different component numbers. Then the total running time of all the feasible problems generated are compared for the two algorithms. Since the running time for 50 and 100 components are quite long, in order to compare, we only generate 1000 feasible problems for each of them.

As we can see, the running time of Hahm & Yano's heuristic is always less than the running time of Jensen's algorithm. For small number of components (up to 5), the difference is tiny. When the number of components becomes larger and larger, the difference between the running time becomes more and more profound. For example, the running time of 30 components for Jensen's algorithm is around 16 times as much as the running time of Hahm & Yano's heuristic.

J	No. of feasible problems generated[1]	Hahm & Yano's algorithm[2]	Jensen's algorithm[2]
2	100000	0.135829	0.162737
3	100000	0.216468	0.263303
5	100000	0.445694	0.663645
7	100000	0.747906	1.44661
10	100000	1.26661	3.86969
20	100000	3.79881	31.5843
30	100000	7.46478	117.925
50	1000	0.191356	6.80933
100	1000	0.753344	75.9417

Table 6.11: Running time compared using the new generation rules.

We have compared the running time of both algorithms using the new generation rules. What about the running time if we use Jensen's generation rules to generate the same number of feasible problems? Will the running time will be the same by different generation rules? Through experiments, we get the Table 6.12.

[1]The feasible problems are generated using the new generation rules.
[2]The running time (in seconds) is a summation of all feasible problems generated.

J	No. of feasible problems generated	Hahm & Yano's algorithm[2]	Jensen's algorithm[2]
2	100000	0.140016	0.16627
3	100000	0.225161	0.248034
5	100000	0.475869	0.513634
7	100000	0.755899	0.874071
10	100000	1.28975	1.61751
20	1000	0.0369404	0.0761266
20	1000[1]	0.037777	0.321867

Table 6.12: Running time compared using Jensen's generation rules.

Still we generate 100,000 feasible problems for different number of components in Table 6.12, but this time the feasible problems are generated using Jensen's generation rules. Because it is difficult to generate feasible problems for the number of components bigger than 10 by Jensen's generator (see Table 6.2), only 1000 feasible problems are generated for 20 components. If we compare Table 6.12 and Table 6.11, we can find that the running time of Jensen's algorithm is different for different generation rules. For 10 components, the running time by the new generator is more than 2 times larger than the running time by Jensen's generator. The reason can be explained in Table 6.13.

J	No. of feasible problems generated by the new generator	Average value of **n**	No. of feasible problems generated by Jensen's generator	Average value of **n**
2	100000	2.11919	100000	2.10399
3	100000	2.42677	100000	2.30968
5	100000	3.27946	100000	2.59612
7	100000	4.29918	100000	2.67482
10	100000	6.03616	100000	2.70535
20	100000	13.7488		
30	100000	24.0518		

Table 6.13: n values compared using both generation rules.

The **n** in Table 6.13 is the size of V (see Table 5.2). The **n** decides the running time in **Step 5**, which decides the overall running time used by Jensen's algorithm. From Table 6.13 we can see that during solving problems generated by Jensen's generation rules, the average value of **n** increases slowly with the number of components increasing. While during solving problems generated using the new generation rules, the average value of **n** increases rapidly with the number of components increasing. For the same number of components, e.g. 10 components, the average value of **n** in the new generator is more than 2 times large than the average value of **n** in Jensen's generator that means the overall running time by solving problems generated using the new rules should be more than 2 times longer than the running time by solving problems generated using Jensen's generator. This is tested and verified by comparing Table 6.11 and Table 6.12.

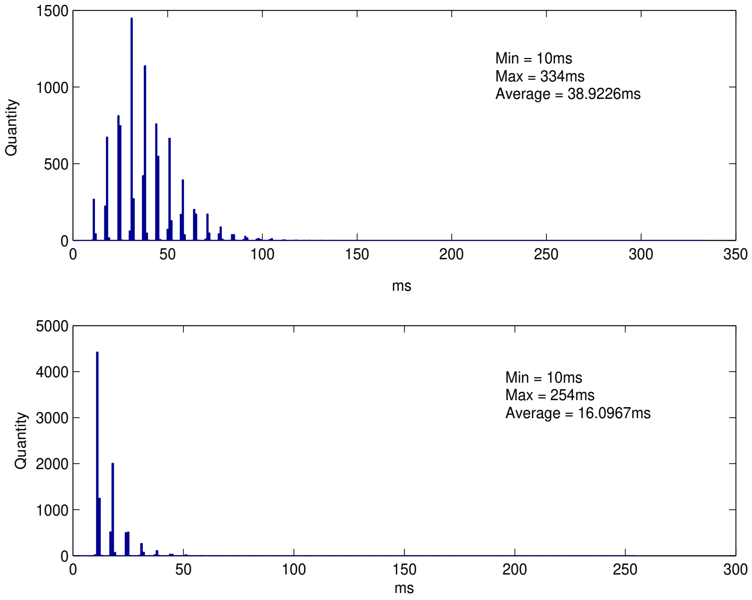

Figure 6.1: Statistics of running time by different generation rules.

Now consider Figure 6.1, we generate 10,000 feasible problems for the number of components 10 using both Jensen's generation rues and the new generation rules. The upper figure is the statistics of different running time of

Jensen's algorithm by solving 10,000 feasible problems using the new rules, and the lower figure uses Jensen's generator. The running time in lower figure are mainly located between 10 and 20 microseconds for each problem, while the running time in the upper figure are mainly located between 10 and 80 microseconds. Obviously the new generator are likely to generate problems with longer running time than Jensen's generator because of the drawing ranges are enlarged with the number of components increasing. And this again leads to the difference between Table 6.11 and Table 6.12.

6.4 Complexity Revisited

The theoretical complexity of Jensen's algorithm has been derived at the end of Chapter 5. The running time of randomly generated problems for different components have been tested. Now we test and verify the correctness of running time both in theory and in practice. The results are showed in Figure 6.2.

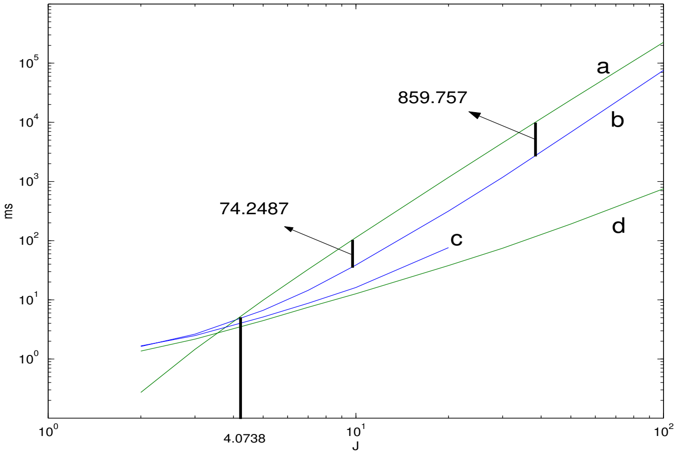

Figure 6.2: Running time comparison by different algorithms.

Curve **a**: theoretical curve of Jensen's algorithm. $a = \mu \times (J^3 \log J)$, where $\mu = 1$.
Curve **b**: running time of Jensen's algorithm: problems generated using the new generation rules.
Curve **c**: running time of Jensen's algorithm: problems generated using Jensen's generation rules.
Curve **d**: running time of Hahm and Yano's algorithm: problems generated using the new generation rules.

In Figure 6.2, the function value of curve **a** is always greater than the function values of curve **b** and **c** when the number of components J is bigger than 4.0738. So the complexity of Jensen's algorithm is verified to be correct.

Curve **b** looks like convergent to curve **a** when the number of components

increases, but in fact they are not. Because the figure is drawn using logarithmic scales for both the X and Y axes. For 10 components, the difference between **a** and **b** is 74.2487. For 20 components, the difference becomes 859.757, which means **a** and **b** are in fact diverging with the number of components increasing.

Curve **b** shows longer running time than curve **c**, and both curve **b** and curve **c** always have longer running time than curve **d**, these have been showed and explained in section 6.3.

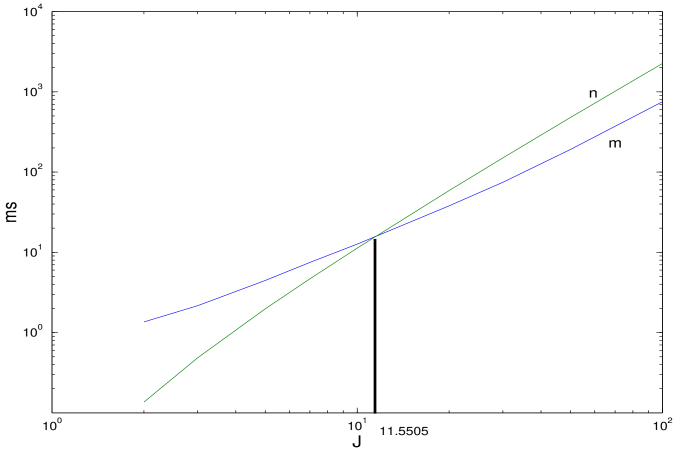

Figure 6.3: The theoretical measure of the execution of Hahm and Yano's algorithm.

Curve **n**: $n = \mu \times (J^2 \log J)$, where $\mu = 1$.
Curve **m**: running time of Hahm and Yano's algorithm: problems generated using the new generation rules ($m = d$ in Figure 6.2).

At the end of Chapter 5 we estimated the complexity of Hahm and Yano's algorithm using the worst case, which is the same as the complexity of Jensen's algorithm $O(J^3 \log J)$. But in fact the running time of Hahm and Yano's algorithm is always shorter than Jensen's algorithm, especially for big problem size. We re-estimate the complexity by drawing Figure 6.3. In Figure 6.3, the curve **m** and curve **n** intersect at the point $J = 11.5505$. we can say this is a theoretical measure of the execution of Hahm and Yano's algorithm. So the complexity of Hahm and Yano's algorithm is $O(J^2 \log J)$ with the number of iterations $O(J)$.

6.5 Statistics of Local Optimum

In order to calculate the number of problems that makes Hahm & Yano's algorithm trap into a local minimum for a given problem size, experiments are tried out using different values of F. The results are listed in Table 6.14. A few selected values of F for every problem size are reported in the table. In the Table 6.14,

Average difference of TC =

$$\frac{\sum_{i=1}^{N} \frac{TC[i]-TC_{best}[i]}{TC_{best}[i]}}{N}$$

Maximal difference of TC =

$$Max\left(\frac{TC[1]-TC_{best}[1]}{TC_{best}[1]}, \frac{TC[2]-TC_{best}[2]}{TC_{best}[2]}, \ldots, \frac{TC[N]-TC_{best}[N]}{TC_{best}[N]}\right)$$

where

TC: the optimal solution of the objective function got from Hahm & Yano's algorithm.
TC_{best}: the optimal solution of the objective function got from Jensen's algorithm.
N: number of problems that Hahm & Yano's algorithm fails.

The experiment reveals that the selection of F decides the number of failures of Hahm & Yano's algorithm. If we increase the value of F, the number of failures will be more than the number of failures with smaller values of F for the same number of feasible problems given. This can be seen in Table 6.14. Because using big values of F to generate problem parameters for the same problem size makes s_r is extremely big and p_r extremely small, which leads to the inequalities (6.2) and (6.3) are more easier to hold, as a result the number of failures of Hahm and Yano's heuristic becomes more with F increasing.

J	No. of feasible problems: new rules	F	No. of failures	Maximal difference of TC	Average difference of TC
2	100000	1	17	0.0211%	<0.01%
		10	534	0.8969%	0.1869%
		60	617	**1.1326%**	**0.2492%**
3	100000	1	67	0.0835%	<0.01%
		5	595	0.4781%	0.0819%
		30	656	1.0377%	0.1663%
5	100000	1	203	0.0342%	<0.01%
		2	469	0.1908%	0.0155%
		20	512	0.5014%	0.0615%
7	100000	1	282	0.0319%	<0.01%
		10	523	0.2020%	0.0236%
10	100000	1	363	0.0199%	<0.01%
		5	533	0.1106%	<0.01%
20	100000	1	339	<0.01%	<0.01%
		3	412	0.0181%	<0.01%
30	100000	1	254	<0.01%	<0.01%
		2	317	<0.01%	<0.01%
50	1000	1	2	<0.01%	<0.01%
100	1000	1	1	<0.01%	<0.01%

Table 6.14: Statistics of the number of failures of Hahm & Yano's algorithm and the difference of TC.

As indicated in Table 6.14, it is quite hard to generate problems for which Hahm and Yano's algorithm fails, and when it fails the difference is tiny. The maximal value of average difference of TC happens at $J = 2$, but it is less than one percent regardless of the setting of F.

6.6 Problem Size Limitation

In previous sections, we have tested problems with problem sizes ranging from 2 to 100. What if the number of components, J, is bigger than 100? The performance is tested in Table 6.15. In Table 6.15, only one problem is generated for each problem size, and we can see that the running time of

Hahm & Yano's algorithm is less than 1 second even for 2000 components. While the running time of Jensen's algorithm is around 1 hour, which is extremely huge.

J	No. of feasible problems: new rules	Hahm & Yano's algorithm (in seconds)	Jensen's algorithm (in seconds)
200	1	0.002	0.94059
500	1	0.018684	24.9429
1000	1	0.072996	291.479
2000	1	0.383327	3563.33(≈1 hour)
5000	1	3.58875	39104.5(≈11 hour)
10000	1	5.00272	> 11 hours

Table 6.15: Running time for problem sizes bigger than 100.

Chapter 7

Conclusion

In order to solve the ELDSP problem in the CCC case, two algorithms are studied and implemented in C++: Hahm & Yano's heuristic algorithm and Jensen's polynomial time algorithm. The reason that leads Hahm & Yano's algorithm fail in some situations is analyzed. Previous experiments are focused on small problem sizes with up to 9, in this report experiments are done using different problem sizes ranging from 2 to 100. Through experiments, we find that the way of generating problems by Jensen is not acceptable, because it can not generate feasible problems for large problem sizes though the experiments did by Jensen are only for small number of components. We come out with a new generator based on Jensen's generation rules to enlarge the drawing ranges while the problem sizes increasing, and the new generation rules work quite well especially for large problem sizes.

The running time of each algorithm for the feasible problems generated are tested and compared. The testing results reveal that the running time of Hahm and Yano's algorithm is quicker than the running time of Jensen's algorithm all the time, the difference is even profound for large problem sizes. Though sometimes Hahm and Yano's algorithm fails to obtain the global optimum, the difference to the globally optimal solution is extremely small. Furthermore, the complexity of Hahm and Yano's algorithm is estimated through experiments.

In fact the problem sizes can not be more than 30 in practice, but the experiments give a deep insight of both algorithms not only in the solution

quality but also in the solution time. As a future work, it is recommended to improve the solution time of Jensen's algorithm in a more efficient way.

Abbreviations & Terminologies

These abbreviations & terminologies are shown in **boldface** the first time they appear.

Average Inventory Level The area below the inventory level curve in a cycle time divided by the cycle time.

CCC The Common Cycle Case.

Earliness The time between the start of production (after setup) of a product and the time of delivery.

ELDSP The Economic Lot and Delivery Scheduling Problem.

EOQ The Economic Order Quantity model.

EPQ The Economic Production Quantity model.

JIT Just In Time; Components are delivered from the supplier immediately before they are needed in the assembly facility.

Uniform Distribution A frequency distribution in which a variable is equally likely to take any value between known limits.

Bibliography

[1] Baker, Kenneth R. (1974) *Introduction to Sequencing and Scheduling*, John Wiley & Sons, Inc. New York. pp. 20-22.

[2] Dobson, G. (1987) The economic lot scheduling problem: achieving feasibility using time-varying lot sizes. *Operations Research*, **35**(5), 764-771.

[3] Gallego, G. and Shaw, D.X. (1997) Complexity of the ELSP with general cyclic schedules *IIE Transactions*, **29**(2), 109-113.

[4] Garey, M.R. and Johnson, D.S. (1979) *Computers and intractability. A guide to the theory of NP-completeness*, W H Freeman & Co., San Francisco.

[5] Hahm, J. and Yano, C.A. (1992) The economic lot and delivery scheduling problem: the single item case. *Internatinal Journal of Production Economics*, **28**, 235-252.

[6] Hahm, J. and Yano, C.A. (1995) The economic lot and delivery scheduling problem: the common cycle case. *IIE Transactions*, **27**, 113-125.

[7] Hahm, J. and Yano, C.A. (1995) The economic lot and delivery scheduling problem: models for nested schedules. *IIE Transactions*, **27**, 126-139.

[8] Hahm, J. and Yano, C.A. (1995) The economic lot and delivery scheduling problem: powers of two policies. *Transportation Science*, **29**(3), 222-241.

[9] Hall, Randolph W. (1996) On the integration of production and distribution: economic order and production quantity implications. *Transportation Research Part B: Methodological*, **30**(5), 387-403.

[10] Hanssmann, F. (1962) *Operations Research in Production and Inventory Control*, John Wiley & Sons, New York, pp. 158-160.

[11] Khouja, M. (2000) The economic lot and delivery scheduling problem:

common cycle, rework, and variable production rate. *IIE Transactions*, **32**(8), 715-725.

[12] Khouja, M. (1997) The scheduling of economic lot sizes on volume flexible production systems. *International Journal of Production Economics*, **48**(1), 73-86.

[13] Khoury, B.N., Abboud, N.E. and Tannous, M.M. (2001) The common cycle approach to the ELSP problem with insufficient capacity. *International Journal of Production Economics*, **73**(2), 189-199.

[14] Mikkel T. Jensen (2001) *Robust and Flexible Scheduling with Evolutionary Computation, PhD Dissertation*, Department of Computer Science, University of Aarhus, Denmark, pp. 197-210.

[15] Wolsey, L.A. (1998) *Integer Programming*, John Wiley & Sons, Inc.

[16] Yu, Gang (1997) Robust economic order quantity models. *European Journal of Operational Research*, **100**(3), 482-493.

Appendix A

Derivations of Formulas

A.1 An alternative way of deriving Q_{opt}

The optimal value of Q in the EOQ model can be alternatively determined in another way.

We can differentiate the objective function with respect to Q, set the resulting function equal to zero (an optimal value can be found as the first derivative is equal to zero), and solve for Q:

$$\begin{aligned} TC &= \frac{C_o D}{Q} + \frac{C_h Q}{2} \\ \frac{\partial TC}{\partial Q} &= -\frac{C_o D}{Q^2} + \frac{C_h}{2} = 0 \\ Q_{opt} &= \sqrt{\frac{2C_o D}{C_h}} \end{aligned}$$

The result is the same as we did in Chapter 2.

A.2 Comparison of αT with the EPQ Model

We know that:

$$\alpha T = \frac{1}{2} T \sum_{i=1}^{J} D_i h_i (1 - p_i D_i)$$

According to the equation (2.2) in the EPQ model in Chapter 2, $p_i D_i$ should be equal to:

$$\frac{d}{p} = \frac{\text{Demand rate of component } i}{\text{Production rate of component } i}$$

This can be approved, as follows:

$$\begin{aligned} \frac{\text{Demand rate of component } i}{\text{Production rate of component } i} &= \frac{\frac{\text{Demand in a cycle time } T}{\text{The cycle time: } T}}{\frac{\text{Demand in a cycle time } T}{\text{Production time in a cycle time } T}} \\ &= \frac{\frac{D_i T}{T}}{\frac{D_i T}{p_i D_i T}} \\ &= \frac{D_i}{\frac{1}{p_i}} \\ &= p_i D_i \end{aligned}$$

A.3 Derivation of formula (6.1)

Requiring (5.2) to hold for all $q \in C_1$, then we can solve equation $TC(T_h, q_g) = TC(T_h, q)$ using the objective function 4.3:

$$
\begin{aligned}
& TC(T_h, q_g) = TC(T_h, q) \\
\Longrightarrow\ & \frac{S+A}{T_h} + \alpha T_h + Z_1(q_g) + Z_2(q_g) \cdot T_h + \beta T_h \\
& \quad = \frac{S+A}{T_h} + \alpha T_h + Z_1(q) + Z_2(q) \cdot T_h + \beta T_h \\
\Longrightarrow\ & Z_1(q_g) + Z_2(q_g) \cdot T_h = Z_1(q) + Z_2(q) \cdot T_h \\
\Longrightarrow\ & T_h = \frac{Z_1(q_g) - Z_1(q)}{Z_2(q) - Z_2(q_g)}
\end{aligned}
$$

And the inequality $T^*(q_g) < T^*(q)$ in 5.2 can be solved using equation (4.4):

$$
\begin{aligned}
& T^*(q_g) < T^*(q) \\
\Longrightarrow\ & \sqrt{\frac{S+A}{\alpha + \beta + Z_2(q_g)}} < \sqrt{\frac{S+A}{\alpha + \beta + Z_2(q)}}
\end{aligned}
$$

So the inequality $T^*(q_g) < T_h < T^*(q)$ becomes:

$$
\sqrt{\frac{S+A}{\alpha + \beta + Z_2(q_g)}} < \frac{Z_1(q_g) - Z_1(q)}{Z_2(q) - Z_2(q_g)} < \sqrt{\frac{S+A}{\alpha + \beta + Z_2(q)}}
$$

which is equal to formula (6.1).

Appendix B

The C++ Codes

The source codes are also available in the following directory at IMM:

`/homestudent/proj/proj43/Thesis/C++/`

B.1 Hahm and Yano's Algorithm

B.1.1 Yano.C

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

#define max(a,b) (((a)>(b))?(a):(b))
#define min(a,b) (((a)<(b))?(a):(b))
#define size 7 //Problem size
#define row  100

#include "sort.h"
#include "print.h"
#include "Z.h"
#include "cond.h"
#include "minA.h"
#include "sum.h"
#include "generator.h"

const int F=1; //Value of F
double S[size],s[size],p[size],h[size],D[size],AA[1],A;

double T,p_h[size],part[size],alpha,beta,SS,ss,pD;
double tau_min,tau_max,Z1,Z2,TC,T_star;
```



```
int flag,n,i,q[row][size];


main() {

  //NGeneral(S,s,p,h,D,AA,F);
  //general(S,s,p,h,D,AA,F);
  special(S,s,p,h,D,AA,F);
  //JSpecial(S,s,p,h,D,AA,F);
  A=AA[0];

  /*
  print(S,size);
  print(s,size);
  print(p,size);
  print(h,size);
  print(D,size);
  */

  n = 0;
  for (i = 0; i < size; i++ )
    q[n][i] = i;

  alpha=0,beta=0,SS=0,ss=0,pD=0;

  for ( i = 0; i < size; i++ ) {
    p_h [i] = p[i]/h[i];
    ss      = ss + s[i];
    pD      = pD + p[i]*D[i];
    alpha   = alpha + 0.5*D[i]*h[i]*(1 - p[i]*D[i]);
    beta    = beta + D[i]*D[i]*p[i]*h[i];
    SS      = SS + S[i];
  }

  tau_min = ss / ( 1 - pD );

  sort( p_h, q, n, 0, size-1 );

  do {

    Z2 = Z(q,n,D,h,p,2);
    T_star = sqrt( (SS+A)/(alpha+beta+Z2) );

    n = n + 1;

    for ( i = 0; i < size; i++ )
      q[n][i] = i;

    T = max( tau_min, T_star );

    for ( i = 0; i < size; i++ ) {
      part[i] = ( T*D[*(*(q+n)+i)]*p[*(*(q+n)+i)]
                + s[*(*(q+n)+i)] )
                / ( D[*(*(q+n)+i)]*h[*(*(q+n)+i)] );
    }
    sort( part, q, n, 0, size-1 );

    flag=cond(q,n);
```

```
  }
  while ( (flag != 0 ) && (T != tau_min) );

  Z1 = Z(q,n,D,h,s,1);
  Z2 = Z(q,n,D,h,p,2);
  TC = (SS + A) / T + alpha*T + Z1 + Z2*T + beta*T;

  cout << "The final results are: " << endl;
  cout << "TC = " << TC << endl;
  cout << "T = " << T << endl;
  print(q,n,size);

  return 0;
}
```


B.2 Jensen's Algorithm

B.2.1 Jensen.C

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

#define max(a,b) (((a)>(b))?(a):(b))
#define min(a,b) (((a)<(b))?(a):(b))
#define size 7 //Problem size
#define row  100

#include "sort.h"
#include "print.h"
#include "Z.h"
#include "minA.h"
#include "sum.h"
#include "generator.h"

const int F=1; //Value of F
double S[size],s[size],p[size],h[size],D[size],AA[1],A;

double part[size],alpha,beta,SS,ss,pD;
int q[size],q_best[size];
int n, m;
double V[1000000];
double TC_best;
double T_best;
double T,Z1,Z2,tau_min,tau_max;
double T_star,TC_T,TC_Vi,TC_Vi_1;

main() {

  int i,j,ma,cont;

  //NGeneral(S,s,p,h,D,AA,F);
  //general(S,s,p,h,D,AA,F);
  special(S,s,p,h,D,AA,F);
  //JSpecial(S,s,p,h,D,AA,F);
  A = AA[0];


  print(S,size);
  print(s,size);
  print(p,size);
  print(h,size);
  print(D,size);


  for(i=0;i<size;i++){
    q[i]=i;
  }

  alpha=0,beta=0,SS=0,ss=0,pD=0;

  for ( i = 0; i < size; i++ ) {
    ss        = ss + s[i];
```

```
    pD       = pD + p[i]*D[i];
    alpha    = alpha + 0.5*D[i]*h[i]*(1 - p[i]*D[i]);
    beta     = beta + D[i]*D[i]*p[i]*h[i];
    SS       = SS + S[i];
  }

  tau_min = ss / ( 1 - pD );
  tau_max = sqrt ( ( SS + A ) / ( alpha + beta ) );

  n = 0;
  m = 0;
  for ( i = 0; i < size; i++ ) {
    for ( j = i+1; j < size; j++ ) {
      T = ( s[j]*D[i]*h[i] - s[i]*D[j]*h[j] )
          /( D[j]*h[j]*D[i]*p[i] - D[i]*h[i]*D[j]*p[j] );

      if ( T > tau_min && T < tau_max ) {
        V[m]=T;
        m = m + 1;
      }
    }
  }
  V[m]=tau_min;
  m = m + 1;
  V[m]=tau_max;

  sort(V,0,m);

  cont = 0;
  for ( i = 0; i < m; i++ ) {
    if ( V[i] < V[i+1] ) {
      V[cont] = V[i];
      cont = cont + 1;
    }
  }
  V[cont] = V[m];

  TC_best = 10000;

  for ( i = 0; i < cont; i++ ) {
    T = 0.5 * ( V[i] + V[i+1] );

    for (j = 0; j < size; j++ ) {
      part[j] = ( T*D[ q[j] ]*p[ q[j] ] + s[ q[j] ] )
                / ( D[ q[j] ]*h[ q[j] ] );
    }

    sort(part,q,0,size-1);

    Z2 = Z(q,D,h,p,2);
    T_star=sqrt( (SS+A) / (alpha+beta+Z2) );

    Z1 = Z(q,D,h,s,1);

    TC_T = (SS+A)/T_star+alpha*T_star+Z1+Z2*T_star+beta*T_star;

    TC_Vi = (SS+A)/V[i]+alpha*V[i]+Z1+Z2*V[i]+beta*V[i];
```



```
    TC_Vi_1 = (SS+A)/V[i+1]+alpha*V[i+1]+Z1+Z2*V[i+1]+beta*V[i+1];

    if(TC_T<TC_best && (V[i]<T_star && T_star<V[i+1]) ){
      TC_best = TC_T;
      T_best = T_star;
      for (  ma = 0; ma < size; ma++ )
        q_best[ma] = q[ma];
    }

    if( TC_Vi < TC_best ) {
      TC_best = TC_Vi;
      T_best = V[i];
      for ( ma = 0; ma < size; ma++ )
        q_best[ma] = q[ma];
    }

    if( TC_Vi_1 < TC_best ) {
      TC_best = TC_Vi_1;
      T_best = V[i+1];
      for ( ma = 0; ma < size; ma++ )
        q_best[ma] = q[ma];
    }
  }

  cout << "\nTC_best = " << TC_best << "\t T_best  = "
       << T_best << endl;
  cout << "q_best is : ";
  print(q_best,size);

  return 0;
}
```

B.3 Subfunctions

B.3.1 cond.h

```
int cond(int (*q)[size], int n){

  int flag = 0;
  for ( int i = 0; i < size; i++ ) {
    if ( *(*(q+n)+i) == *(*(q+(n-1))+i) )
      flag = flag + 0;
    else
      flag = flag + 1;
  }
  return flag;
}

int cond(int (*q)[size], int *c, int n){

  int flag = 0;
  for ( int i = 0; i < size; i++ ) {
    if ( *(*(q+n)+i) == c[i] )
      flag = flag + 0;
    else
      flag = flag + 1;
  }
  return flag;
}
```

B.3.2 generator.h

```
/* Jensen's generator */
void general(double S[],double s[],double p[],
             double h[],double D[],double AA[],int F)
{
  int i,xmen;

  AA[0] = 5.0*rand()/RAND_MAX;
  for ( i = 0; i < size; i++ ) {
    S[i] = 1.0*rand()/RAND_MAX;
    h[i] = 1.0*rand()/RAND_MAX;
  }

  xmen = 1.0*(size-1)*rand()/RAND_MAX;
  s[xmen] = 0;
  p[xmen] = 100;
  for ( i = 0; i < size; i++){
    if ( i != xmen ) {
      s[i] = 0.25*rand()/RAND_MAX;
      do {
        p[i] = 1.0*rand()/RAND_MAX;
        D[i] = 1.0*rand()/RAND_MAX;
      }
      while( p[i]*D[i] >= 1.0/size );
    }
  }
  s[xmen] = F*sum(s, size);
  p[xmen] = 1.0/F*minA(p,size);
  do {
```



```
    D[xmen] = 1.0*rand()/RAND_MAX;
  }
  while(p[xmen]*D[xmen] >= 1.0/size);
}

/*New generator */
void NGeneral(double S[],double s[],double p[],
              double h[],double D[],double AA[],int F)
{
  int i,xmen;

  AA[0] = 5.0*rand()/RAND_MAX;
  for ( i = 0; i < size; i++ ) {
    S[i] = 1.0*rand()/RAND_MAX*sqrt(size*1.0);
    h[i] = 1.0*rand()/RAND_MAX;
  }

  xmen = 1.0*(size-1)*rand()/RAND_MAX;
  s[xmen] = 0;
  p[xmen] = 100;
  for ( i = 0; i < size; i++){
    if ( i != xmen ) {
      s[i] = 0.25*rand()/RAND_MAX/sqrt(size*1.0);
      do {
        p[i] = 1.0*rand()/RAND_MAX;
        D[i] = 1.0*rand()/RAND_MAX;
      }
      while( p[i]*D[i] >= 1.0/size );
    }
  }
  s[xmen] = F*sum(s, size);
  p[xmen] = 1.0/F*minA(p,size);
  do {
    D[xmen] = 1.0*rand()/RAND_MAX;
  }
  while(p[xmen]*D[xmen] >= 1.0/size);
}

/* New, tau_min < tau_max is confirmed */
void special(double S[],double s[],double p[],
             double h[],double D[],double AA[],int F)
{
  int i,xmen;
  double alpha,beta,SS,ss,pD;
  double tau_min,tau_max;

  do {

    AA[0] = 5.0*rand()/RAND_MAX;
    for ( i = 0; i < size; i++ ) {
      S[i] = 1.0*rand()/RAND_MAX*sqrt(size*1.0);
      h[i] = 1.0*rand()/RAND_MAX;
    }

    xmen = 1.0*(size-1)*rand()/RAND_MAX;
    s[xmen] = 0;
    p[xmen] = 100;
    for ( i = 0; i < size; i++){
```

```
      if ( i != xmen ) {
        s[i] = 0.25*rand()/RAND_MAX/sqrt(size*1.0);
        do {
          p[i] = 1.0*rand()/RAND_MAX;
          D[i] = 1.0*rand()/RAND_MAX;
        }
        while( p[i]*D[i] >= 1.0/size );
      }
    }
    s[xmen] = F*sum(s, size);
    p[xmen] = 1.0/F*minA(p,size);
    do {
      D[xmen] = 1.0*rand()/RAND_MAX;
    }
    while(p[xmen]*D[xmen] >= 1.0/size);

    /******************************************/

    alpha = 0,beta = 0,SS = 0,ss = 0,pD = 0;

    for ( i = 0; i < size; i++ ) {
      ss     = ss + s[i];
      pD     = pD + p[i]*D[i];
      alpha  = alpha + 0.5*D[i]*h[i]*(1 - p[i]*D[i]);
      beta   = beta + D[i]*D[i]*p[i]*h[i];
      SS     = SS + S[i];
    }
    tau_min = ss / ( 1 - pD );

    tau_max = sqrt ( ( SS + AA[0] ) / ( alpha + beta ) );

  }
  while(tau_min>=tau_max);

}

/* Jensen, tau_min < tau_max is confirmed */
void JSpecial(double S[],double s[],double p[],
              double h[],double D[],double AA[],int F)
{
  int i,xmen;
  double alpha,beta,SS,ss,pD;
  double tau_min,tau_max;

  do {

    AA[0] = 5.0*rand()/RAND_MAX;
    for ( i = 0; i < size; i++ ) {
      S[i] = 1.0*rand()/RAND_MAX;
      h[i] = 1.0*rand()/RAND_MAX;
    }

    xmen = 1.0*(size-1)*rand()/RAND_MAX;
    s[xmen] = 0;
    p[xmen] = 100;
    for ( i = 0; i < size; i++) {
      if ( i != xmen ) {
        s[i] = 0.25*rand()/RAND_MAX;
```



```
        do {
          p[i] = 1.0*rand()/RAND_MAX;
          D[i] = 1.0*rand()/RAND_MAX;
        }
        while( p[i]*D[i] >= 1.0/size );
      }
    }
    s[xmen] = F*sum(s, size);
    p[xmen] = 1.0/F*minA(p,size);
    do {
      D[xmen] = 1.0*rand()/RAND_MAX;
    }
    while(p[xmen]*D[xmen] >= 1.0/size);

    /******************************************/

    alpha = 0,beta = 0,SS = 0,ss = 0,pD = 0;

    for ( i = 0; i < size; i++ ) {
      ss     = ss + s[i];
      pD     = pD + p[i]*D[i];
      alpha  = alpha + 0.5*D[i]*h[i]*(1 - p[i]*D[i]);
      beta   = beta + D[i]*D[i]*p[i]*h[i];
      SS     = SS + S[i];
    }
    tau_min = ss / ( 1 - pD );

    tau_max = sqrt ( ( SS + AA[0] ) / ( alpha + beta ) );

  }
  while(tau_min>=tau_max);

}
```

B.3.3 maxA.h

```
double maxA (double *ia, int sz) {
  double temp=0;
  for ( int i = 0; i < sz; i++ ) {
    temp = max(ia[i],temp);
  }
  return temp;
}
```

B.3.4 minA.h

```
double minA (double *ia, int sz) {
  double temp = 100;
  for ( int i = 0; i < sz; i++ ) {
    temp = min ( ia[i], temp );
  }
  return temp;
}
```

B.3.5 print.h

```
const int lineLength = 10; // elements in a line
```

```
void print( double *ia, int sz) {

  cout<< "double ["
      << sz
      << "]={";

  for (int i = 0; i < sz; ++i) {
    if ( i % lineLength == 0 && i)
      cout << "\n\t"; // line filled

    cout << ia[i]
         << ",";
  }
  cout << "};\n";
}

void print( int (*ia)[size], int n, int sz) {

  cout << "("
       << sz
       << "): {";

  for (int i = 0; i < sz; ++i) {
    if ( i % lineLength == 0 && i)
      cout << "\n\t"; // line filled

    cout << *(*(ia+n)+i)
         << ",";
  }
  cout << "}\n";
}

void print( int *ia, int sz) {

  cout << "["
       << sz
       << "]={";

  for (int i = 0; i < sz; ++i) {
    if ( i % lineLength == 0 && i)
      cout << "\n\t"; // line filled

    cout << ia[i];

    // separate all but last element
    //if ( i % lineLength != lineLength-1 && i != sz-1 )
      cout << ",";
  }
  cout << "}\n";
}
```

B.3.6 sort.h

```
void swap ( double *ia, int i, int j ) {
  double tmp = ia[i];
       ia[i] = ia[j];
       ia[j] = tmp;
}
```



```
//sorts an array in non-increasing order.
void sort( double *ia, int *q, int low, int high ) {
  // stopping condition for recursion
  if(low<high) {
    int lo = low;
    int hi = high + 1;
    double elem = ia[low];

    for ( ; ; ) {
      while ( ia[++lo] > elem );
      while ( ia[--hi] < elem );

      if ( lo < hi ) {
        swap ( ia, lo, hi );
        int temp = q[lo];
        q[lo] = q[hi];
        q[hi] = temp;
      }
      else break;
    }   //end, for(;;)

    swap( ia, low, hi );
    int temp = q[low];
    q[low] = q[hi];
    q[hi] = temp;

    sort( ia, q, low, hi - 1 );
    sort( ia, q, hi + 1, high );
  }     //end, if( low < high)
}

void sort( double *ia, int (*q)[size], int n, int low, int high ) {
  // stopping condition for recursion
  if(low<high) {
    int lo = low;
    int hi = high + 1;
    double elem = ia[low];

    for ( ; ; ) {
      while ( ia[++lo] > elem );
      while ( ia[--hi] < elem );

      if ( lo < hi ) {
        swap ( ia, lo, hi );
        int temp = *(*(q+n)+lo);
        *(*(q+n)+lo) = *(*(q+n)+hi);
        *(*(q+n)+hi) = temp;
      }
      else break;
    }   //end, for(;;)


    swap( ia, low, hi );
    int temp = *(*(q+n)+low);
    *(*(q+n)+low) = *(*(q+n)+hi);
    *(*(q+n)+hi) = temp;
    sort( ia, q, n, low, hi - 1 );
    sort( ia, q, n, hi + 1, high );
```

```
  }      //end, if( low < high)
}

// qsort sorts an array in increasing order.
void sort( double *ia, int low, int high ) {
  // stopping condition for recursion
  if(low<high) {
    int lo = low;
    int hi = high + 1;
    double elem = ia[low];

    for ( ; ; ) {
      while ( ia[++lo] < elem );
      while ( ia[--hi] > elem );

      if ( lo < hi ) {
        swap ( ia, lo, hi );
      }
      else break;
    }   //end, for(;;)

    swap( ia, low, hi );
    sort( ia, low, hi - 1 );
    sort( ia, hi + 1, high );
  }      //end, if( low < high)
}
```

B.3.7 sum.h

```
double sum (double *ia, int sz) {
  double temp = 0;
  for ( int i = 0; i < sz; i++ ) {
    temp = temp + ia[i];
  }
  return temp;
}

double sum (int *ia, int sz) {
  int temp = 0;
  for ( int i = 0; i < sz; i++ ) {
    temp = temp + ia[i];
  }
  return temp;
}

double sumD (double *ia, double *ib, int sz) {
  double temp = 0;
  for ( int i = 0; i < sz; i++ ) {
    temp = temp + ia[i]*ib[i];
  }
  return temp;
}
```

B.3.8 Z.h

```
double Z(int *q, double *D, double *h, double *x,int d) {

  if(d==1) {
    double Z1 = 0;
```



```
    for ( int i = 0; i < size; i++ ) {
      for ( int j = i+1; j < size; j++ ) {
        Z1 = Z1 + D[q[i]]*h[q[i]]*x[q[j]];
      }
    }
    return Z1;
  }
  else {
    double Z2 = 0;
    for ( int i = 0; i < size; i++ ) {
      for ( int j = i+1; j < size; j++ ) {
        Z2 = Z2 + D[q[i]]*h[q[i]]*D[q[j]]*x[q[j]];
      }
    }
    return Z2;
  }
}

double Z(int (*q)[size],int n,double *D,double *h,double *x,int d){

  if(d==1) {
    double Z1 = 0;
    for ( int i = 0; i < size; i++ ) {
      for ( int j = i+1; j < size; j++ ) {
        Z1 = Z1 + D[*(*(q+n)+i)]*h[*(*(q+n)+i)]*x[*(*(q+n)+j)];
      }
    }
    return Z1;
  }
  else {
    double Z2 = 0;
    for ( int i = 0; i < size; i++ ) {
      for ( int j = i+1; j < size; j++ ) {
        Z2 = Z2 + D[*(*(q+n)+i)]*h[*(*(q+n)+i)]
             *D[*(*(q+n)+j)]*x[*(*(q+n)+j)];
      }
    }
    return Z2;
  }
}
```