Modelling a Complex Production Scheduling Problem
- Optimization Techniques

Emmanuel Sohier
This thesis is submitted to the Department of Interaction and System Design, School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author:
Emmanuel Sohier
E-mail: sohieremmanuel@hotmail.com

University advisor:
Jan Persson
Department of Systems and Software Engineering

Department of Interaction and System Design
Blekinge Institute of Technology
Box 520
SE 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : + 46 457 102 45
Sweden
Abstract

In this thesis, a complex real-world problem, a sequence dependent scheduling of different product orders on a number of lines is addressed. Changeover costs occur between product orders belonging to different product groups. The operational research cycle is employed exploring different optimization techniques such as mathematical modelling and heuristic approaches. The identification, implementation and demonstration of the techniques are supported with numerical results from experiments. One combination of different solution techniques is put forward. Some suggestions are done for reducing cost and increasing productivity.

Keywords: real scheduling, optimization techniques, operational research
# Contents

## 1 Introduction

1.1 Background ................................................................. 3  
1.1.1 Challenge ............................................................... 3  
1.1.2 Purpose ................................................................. 3  
1.1.3 Research Questions .................................................. 3  
1.2 Problem type and models ................................................. 4  
1.3 Solution Techniques .................................................... 4  
1.4 Method ........................................................................... 4  
1.5 Outline ........................................................................... 5  

## 2 Problem Description and Model

2.1 Problem Description ......................................................... 6  
2.2 The Mathematical Model .................................................. 7  
2.2.1 Introduction ............................................................... 7  
2.2.2 Description ............................................................... 8  
2.2.3 Specification ............................................................. 9  
2.2.4 Remarks ................................................................... 11  

## 3 Solution Techniques

3.1 Using a Modelling Language - AMPL ................................... 12  
3.2 Representation of Solution (Decoding and Encoding) ............ 12  
3.2.1 The Evaluation Function ............................................. 13  
3.3 Genetic Algorithm (GA) ................................................... 14  
3.3.1 Different Crossover Mechanisms ................................. 14  
3.3.2 Specification ............................................................ 15  
3.4 Tabu Search ................................................................. 16  
3.4.1 Specification ............................................................. 17  
3.5 Pairwise Change ............................................................ 17  
3.6 A Combined Approach: Genetic Algorithm and Pairwise Change .... 18  

## 4 Computational Results

4.1 AMPL Results ................................................................... 19  
4.2 Input Data ..................................................................... 19  
4.3 Setting .......................................................................... 20  
4.4 Genetic Algorithm: Greedy Subtour Crossover versus Position Based Crossover .................. 20  
4.5 Improving Local Search: Full versus Line Pairwise Change .. 21  
4.6 Tabu Search Approach ...................................................... 22  
4.7 A Combined Approach: Genetic Algorithm and Pairwise Change ........................................... 22  
4.8 Adding CPU-time to a Combined Approach ....................... 22  
4.9 Computational Comparison .............................................. 22  
4.10 Cost of Inflexibility ......................................................... 22
Chapter 1

Introduction

1.1 Background

1.1.1 Challenge

The thesis searches for models and algorithms to provide an efficient support for production scheduling and its analysis.

The complexity of a real world factory problem makes it a difficult problem. It is impossible to map all cause-effect relationships in the studied system.

The scheduling problem studied is complex; there is probably no algorithm able to solve it in polynomial time.

The presence of constraints on the different resources make real scheduling problems complicated.

The thesis presents a suitable model and examines different solution techniques within optimization. The thesis suggests opportunities to lower costs.

1.1.2 Purpose

Scheduling and sequencing play a decisive role in manufacturing and other industries, it is essential for survival. An example of the complex scheduling routine is studied at AarhusKarlshamn, where this routine is currently mainly done manually by experienced planners.

Research of complex processes in general can increase the efficiency in many different areas.

Therefore, the purpose of this research is to evaluate different solution techniques and uses optimization techniques to potentially assist in possible improvements in other connected activities.

1.1.3 Research Questions

What is a good model and solution technique suitable for addressing the following multi-purpose usage:

- Assist in operational scheduling.
- Improve different policies, aid in the education of planners.
- Support decomposition approaches, find hereby opportunities to reduce cost in other connected activities. It is outlined in [Persson and Davidsson(2005)] that, in order to support decomposition, the algorithm needs to be fast and, dependent on the specific problem decomposition, able to handle different types of costs.
- Identification of possible consequences of change in demand.
The "assist in operational scheduling" usage is mainly addressed in this thesis, however the other usages are reflected in the selection of certain alternatives.

1.2 Problem type and models

Production management problems can be split up into different categories. We can differentiate between production planning, production scheduling, and production control. Production planning is the highest level. A decision at this level could be to decide for the total amount to produce in the next quarter. In this level capacity constraints can be dealt with as variables. Production scheduling has a shorter time horizon. The allocation of the different resources is done in this level. Production control, the lowest level, is a real time task. This level makes sure that the planning produced in upper levels is carried out. [Brandimarte and Villa(1995)]

The discussed problem is situated in the production planning level, reaching into the production control level.

We can distinguish the following three levels of decision making: strategic decision making; dealing with long-term implications of decisions, tactical decision making: deals with the medium-term implications, and operational decision making dealing with short-term implications. We can categorize the addressed scheduling problem in the operational and tactical decision making level, however the research could also assist in tackling the higher level.

There are different classes of optimization models. One can divide into continuous and discrete problems. The presented model is a continuous time problem but has decision variable constrained to discrete values.

1.3 Solution Techniques

Optimization involves formulating the problem in terms of decision variables. It searches for a maximum or a minimum of objective function(s) expressed in terms of the decision variables. The variables are limited to certain values expressing limits on the decision choices. [Rardin(2000)]

Optimization techniques cover a wide area of techniques: mathematical modelling, linear programming, improving search, discrete optimization methods, etc.

Modelling has been referred to as "the art of selective simplification of reality". Operations Research explores best ways to model mathematically different kinds of complex problems. It also studies the analysis of the formed model regarding possible solutions to the problems. Operations Research applications were successful in different settings. Many different approaches exist to tackle this complex scheduling problem. There is the choice between simulation, a descriptive way of modelling, and the prescriptive modelling path; this entails a trade-off between validity and tractability. Another choice is exact versus heuristic optimization. Exact implies that the suggested solution is guaranteed to be as good as any other possible solution. There is the difference between deterministic and stochastic models. Stochastic means that there are probabilities involved in the input data.[Rardin(2000)]

This research touches both descriptive and prescriptive modelling, exact and heuristic optimization, and stochastic models.

1.4 Method

The Operations Research cycle approach [Rardin(2000)] has been followed in this research. This cycle (Figure 1.1 on the following page) starts with the formulation of the problem. A rough modelling of the problem along the available information follows. The different relationships in the system and the different parameters are quantified. The suggested mathematical model is implemented to obtain conclusions. These conclusions purely come from the model, not from
the real problem. What follows is interference. The conclusions are validated with a person who has a deeper knowledge of the actual problem. This leads to detection of strong and weak points in the model. This is translated in changes in the model to obtain more usability. The described loop is iterated several times. Operations Research techniques have proved their worth in very different application settings.

![Operations Research Process](image)

Figure 1.1: Operations Research Process [Rardin(2000)]

1.5 Outline

We start off describing the real scheduling problem at AarhusKarlshamn. We give a short introduction before presenting the mathematical model, which we explain more thoroughly afterwards.

The next chapter presents the different possible solution techniques. We explain the used solution representation. The solution techniques are also specified in more detail.

We present the computational results obtained using the discussed solution techniques. The setting of the experiments is introduced prior to presenting their results.

The thesis ends with conclusions and a suggestion of future work.
Chapter 2

Problem Description and Model

2.1 Problem Description

This problem description is based on an interview of Jan Persson with Tommy Berger; an employee of AarhusKarlshamn, an introductory meeting with the planning section of AAK, and a document belonging to the project: "Integrated Production and Transportation Planning within Food Industry"; active on http://www.ipd.bth.se/fatplan/.

AarhusKarlshamn is an international manufacturer of high value-added speciality vegetable oils and fats partly situated in Karlshamn. The last production step (Figure 2.1 on the next page), named DESO, is complex to plan. DESO consists of three production lines with a total daily capacity of 900 tonnes producing around 220 kiloton per year round the clock six days a week.

The three lines have different production capacity per time unit. Provided input to the problem are a number of product orders (including product type, quantities, time of deliver and delivery address). The specification of the product orders are typically available at least two days before their production needs to be finished.

It it necessary to decide when to produce a product order and in which production line. Some lines are better equipped to process a certain product order. Production occurs typically in batches of 5 tonnes or multiples of five tonnes. Up to seven components can be used in one and the same batch.

The planning is difficult because of the presence of changeover costs and effects, e.g. cleaning, called spooling, might be necessary for some sequences of production batches. The viscosity of the product order plays an important part in the stipulation of the changeover effects. Changing should preferably happen from less viscous to more viscous to avoid certain solidify effects. Cases where changeover costs apply are rendered with a "C" in Table 2.1 on the following page. The production batches need to be planned early enough, such that loading and transportation with truck can be carried out in time for ensuring delivery on time. On the other hand, batches should not be produced too early because of limited inventory capacities and freshness requirements (products are typically not stored longer than 12 hours). Inventory storage capacities for finished product orders are limited. This implies that product orders are often finished in the same day as their delivery.

There are furthermore certain orders who need to be prioritized, e.g. ship transport orders (to meet the shipping date) and orders with a long transportation. There is also a possibility to use overcapacity of DESO to produce against stock of a bulk product. Some more soft constraints are present in the problem.
Figure 2.1: DESO process

<table>
<thead>
<tr>
<th></th>
<th>Liquid</th>
<th>Non liquid</th>
<th>Non-trans</th>
<th>Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Half-liquid</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Chocolate</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>CBE</td>
<td>C</td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>CBR</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>CBS+C12</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Low trans</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Liquid (LOBRA)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Half-liquid</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>CBE</td>
<td>C</td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>CBR</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>CBS+C12</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Trans</td>
<td>C</td>
<td>(C)</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Other</td>
<td>C</td>
<td>(C)</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

Table 2.1: Changeover Effect

2.2 The Mathematical Model

2.2.1 Introduction

One of the most commonly used models in operational research are mathematical programs. They assist in analyzing and understanding the problem by abstracting the problem to a certain level from its complex real world setting. It is a research area by itself. However the applicability of the model to solve the problem is not always clear in an initial stage.

A mathematical model essentially consists of different mathematical relationships. Such relationships can be equations, inequalities, logical dependencies and so on. These relationships represent relationships from the real world setting. Usually there is one objective function expressed in terms of variables and parameters, and many constraints on the variables. Mathematical programming then describes the minimization or maximization of the objective function subsidiary to the constraints.

The different stages in practical mathematical programming are very similar to the operational research cycle. however specified a more detailed cycle:
• "Formulate a model, the abstract system of variables, objectives, and constraints that represent the general form of the problem to be solved.

• Collect data that define a specific problem instance.

• Generate a specific objective function and constraint equations from the model and data.

• Solve the problem instance by running a program, or solver, to apply an algorithm that finds optimal values of the variables.

• Analyze the results.

• Refine the model and data as necessary, and repeat.”

There are different categories of mathematical programs. A special case is called linear programming, which mathematically entails that the objective function and all constraints are linear equations and inequalities. The advantage of this case is that there exist a lot of fast methods to solve linear programs and the guaranteed optimal solution. In the described model some variables are constraint to integer values. This more complicated case is called integer programming. But greater computational force of computers and development of advanced methods have also made integer programs solvable. The described mathematical model has both integer and linear constraints. We can call it a mixed integer program. The described mathematical model is a single objective problem.

The next section describes the mathematical model, a more thorough explanation follows in the subsequent section.

2.2.2 Description

We use the following notation for sets:
Set P is the set of product orders, denoted by index p
Set L is the set of lines, denoted by index l
Set \( S_l \) is the set of sequences on line l, denoted by index s

Furthermore, parameters are denoted by a common letter, variables by a capital letter.

\( d_p^{time} \): time to finish a product order p
\( c_{\tilde{p},p} \): the cost to change production from product order \( \tilde{p} \) to product order \( p \)
\( f_{p,l} \): time for processing a product order on a line l

\( O_{\tilde{p},p,l,s} \): 1 if there appears a changeover from product order \( \tilde{p} \) to product order \( p \) in line l in sequence s

\( X_{p,l,s} \): 1 if product order p is scheduled in line l in sequence s

\( Y_{p,l,s} \): time in which product order p is finished on line l in sequence s

Minimize the cost:

\[
\min Z = \sum_{\tilde{p} \in P} \sum_{p \in P} \sum_{l \in L} \sum_{s \in S_l} (c_{\tilde{p},p} \cdot O_{\tilde{p},p,l,s})
\]  

(2.1)

Connection time variable:

\[
Y_{p,l,s} \geq f_{p,l} X_{p,l,s} + \sum_{p' \in P} Y_{p',l,s-1} - M(1 - X_{p,l,s}) \quad p \in P, l \in L, s \in S_l
\]  

(2.2)

\[
Y_{p,l,s} \leq M X_{p,l,s} \quad p \in P, l \in L, s \in S_l
\]  

(2.3)
Demand constraint:

$$\sum_{l \in L} \sum_{s \in S_l} Y_{p,l,s} \leq d^\text{time}_p$$

$p \in P$ (2.4)

Connection changeover variable:

$$O_{\overline{p},\overline{p},l,s} \geq X_{\overline{p},l,s} + X_{\overline{p},l,s-1} - 1$$

$p, \overline{p} \in P, l \in L, s \in S_l$ (2.5)

Initial value:

$$Y_{p,l,0} = 0$$

$p \in P, l \in L$ (2.6)

Production:

$$\sum_{l \in L} \sum_{s \in S_l} X_{p,l,s} = 1$$

$p \in P$ (2.7)

Sequencing:

$$\sum_{p \in P} X_{p,l,s} \leq 1$$

$l \in L, s \in S_l$ (2.8)

Production Sequencing:

$$\sum_{p \in P} X_{p,l,s} \leq \sum_{p \in P} X_{p,l,s-1}$$

$l \in L, s \in S_l : s > 1$ (2.9)

Binary constraints:

$$O_{\overline{p},\overline{p},l,s} = 0 \text{ or } 1$$

$p, \overline{p} \in P, l \in L, s \in S_l$ (2.10)

$$X_{p,l,s} = 0 \text{ or } 1$$

$p \in P, l \in L, s \in S_l$ (2.11)

Non-negative constraint:

$$Y_{p,l,s} \geq 0$$

$p \in P, l \in L, s \in S_l$ (2.12)

2.2.3 Specification

Sets, Parameters, and Variables

The sets, parameters, and variables are self describing and do not need an extensive explanation.

$c_{\overline{p},\overline{p}}$ is a $P \times P$ matrix with zeros in its left-to-right diagonal (no changeover costs from same product order to same product order). $f_{p,l}$ is a $P \times L$ matrix stating the processing times of the product orders on a specific line.

$O_{\overline{p},\overline{p},l,s}$ can be seen as a four dimensional matrix. $X_{p,l,s}$ is a three dimensional variable. We present a reduced possible representation of this matrix in Table 2.2 on the next page. $Y_{p,l,s}$ can be represented in a similar table, replacing the ”ones” with an appropriate time value.

Objective Function

The objective function (equation 2.1) is a summation of all changeover costs created during production. The objective is to keep this sum as low as possible.
Constraints

- Equation 2.2 and 2.3 take together with equation 2.12 care of setting \( Y_{p,l,s} \) to its value. We introduce the "sufficiently large" M-constant. We explain the equations by passing through the possible cases.

  - Setting a lower bound value (equation 2.2 and 2.12)
    * \( X_{p,l,s} \) is equal to one (there is production for that particular product order in that certain line and sequence): the time a certain product is finished \( (Y_{p,l,s}) \) has to be greater than its production time \( (f_{p,l}) \) added to the finished time in the previous sequence.
    * \( X_{p,l,s} \) is equal to zero (there is no production for that particular product order in that certain line and sequence): the "sufficiently large" M-constant should be set larger than the largest possible finished production time. This way the lower bound value of \( Y_{p,l,s} \) is set to a negative number. However, the non-negative constraint in 2.12 forces the lower bound value to be greater than or equal to zero.

  - Setting an upper bound value (equation 2.3)
    * \( X_{p,l,s} \) is equal to one: the time a certain product is finished \( (Y_{p,l,s}) \) is pushed to be lower than the largest possible finished production time (M).
    * \( X_{p,l,s} \) is equal to zero: the time a certain product is finished \( (Y_{p,l,s}) \) is pushed to be lower than zero.

- Equation 2.4 formulates mathematically that every production order needs to be produced in time (before or equal to \( d_{p,\text{time}} \)).

- The objective function strives for as many \( o_{p,p',p,l,s} \) values to be zero through the minimization. Equation 2.5 on the other hand forces \( o_{p,p',p,l,s} \) to take a "one value" if and only if there is a changeover between two sequences on a particular line. The mechanism is easy to understand with the help of Table 2.3 on the following page.

- The earlier mentioned variable \( X_{p,l,s} \) represents the sequencing of the product orders on the different lines. Two constraints are:

<table>
<thead>
<tr>
<th>line 1</th>
<th>( X_{p,l,s} )</th>
<th>( \text{po 3} )</th>
<th>( \text{po 4} )</th>
<th>( \text{po 5} )</th>
<th>( \text{po 6} )</th>
<th>( \text{po 7} )</th>
<th>( \text{po 8} )</th>
<th>( \text{po 9} )</th>
<th>( \text{po 10} )</th>
<th>( \text{po 11} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq. 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>seq. 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>seq. 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>line 2</th>
<th>( X_{p,l,s} )</th>
<th>( \text{po 3} )</th>
<th>( \text{po 4} )</th>
<th>( \text{po 5} )</th>
<th>( \text{po 6} )</th>
<th>( \text{po 7} )</th>
<th>( \text{po 8} )</th>
<th>( \text{po 9} )</th>
<th>( \text{po 10} )</th>
<th>( \text{po 11} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq. 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>seq. 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>seq. 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>line 3</th>
<th>( X_{p,l,s} )</th>
<th>( \text{po 3} )</th>
<th>( \text{po 4} )</th>
<th>( \text{po 5} )</th>
<th>( \text{po 6} )</th>
<th>( \text{po 7} )</th>
<th>( \text{po 8} )</th>
<th>( \text{po 9} )</th>
<th>( \text{po 10} )</th>
<th>( \text{po 11} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq. 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>seq. 2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>seq. 3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.2: Possible Representation of \( X_{p,l,s} \)
Table 2.3: Truth Table

<table>
<thead>
<tr>
<th>$X_{p,l,s-1}$</th>
<th>$X_{p,l,s}$</th>
<th>sum $O_{p,p,l,s}(\text{sum} - 1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Each product order has to be processed during a line sequence (equation 2.7). Formulated differently: there has to be one "one value" in each column in the Table 2.2 representation.

Each line can only process one product order in a sequence (equation 2.8). Formulated differently: there can at most be one "one value" in each row in the Table 2.2 representation.

- Equation 2.9 makes sure there can only be production on a particular line and sequence if there was production in the previous sequence on the same line. It rules out option 4 (in italic) in Table 2.3.

- Equations 2.10 and 2.11 make sure $O_{p,p,l,s}$ and $X_{p,l,s}$ are binary variables.

2.2.4 Remarks

There are additional costs considered when using heuristics later in this thesis, which are currently not included in the mathematical model. The cost for late or early production is taken in account in the solution representation presented later where it is not included in the mathematical model. Same can be said on the cost linked to production on a specific line.

It is possible to extend the mathematical model in different ways. It is still a rather general representation of the minimization of changeover costs in the sequenced production planning of product orders on different production lines.
Chapter 3

Solution Techniques

This chapter describes the different solution techniques employed on the problem. The first solution technique presented is the usage of a modelling language and a standard solution approach branch-and-bound. An introduction to the deployed solution representation is given. What follows is an explanation of the genetic algorithm approach, tabu search, pairwise change, and a short note on a combined approach.

3.1 Using a Modelling Language - AMPL

AMPL stands for "A Mathematical Programming Language". It is a modeling language used to develop and apply mathematical programming models. The first edition of AMPL came out in 1992 and the language has evolved ever since.

Implementation in AMPL is a natural first step once the mathematical model is formulated. It is possible to obtain information on the correctness, solvability, usability, and computability of the developed model.

AMPL needs the mathematical programming model and an instance of the data to solve the problem. These are fed into the AMPL program that works as a compiler. AMPL translates them to a readable version for a specified solver. The solver applies an appropriate algorithm to compute an optimum and gives this solution back to the AMPL program. [Gay(1997)] Used solver is CPLEX, which can solve linear programming problems, integer programs, and also quadratic programming. It is considered a state-of-the-art solver for mixed integer problems; and is a good solver to check whether standard software can be used for solving an optimization problem.

The AMPL code is syntactically very closely connected to the mathematical model with parameters, variables, one objective function and many constraints. The code is put in the appendix.

3.2 Representation of Solution (Decoding and Encoding)

The here presented solution representation is known in the Genetic Algorithm jargon as path representation. Other possible representations often used are binary, adjacency, ordinal, and matrix representation.

The path representation is modified to fit the specific problem. A possible solution is represented by an array where the first element of the array is used to contain the function value. Further, the '1' value is a sentinel indicating the end of the first production line and the start of the second production line. The '2' value is a sentinel indicating the end of the second production line, start of the third production line. Each other value corresponds with a production order ranging from 3 to ... We give a simplified example for a better understanding in the next paragraph and Table 3.1.
Table 3.1: Solution Representation

The '0' value is the function value for this particular solution. The example indicates that production on the first line is started with product order 9. Then product orders $5 \rightarrow 12 \rightarrow 8$ are processed. The sentinel value "2" indicates we move on to the third line where production starts with product order $10 \rightarrow 7 \rightarrow 4$. Sentinel value "1" indicates the second line. Production here is in the sequence: $11 \rightarrow 6 \rightarrow 3$.

One advantage of this representation is that it is very easy to generate new starting solutions by randomly shuffling the numbers in the array. Another advantage is that every solution generated is acceptable.

Each time a member needs to be evaluated it is necessary to decode the array of the member.

### 3.2.1 The Evaluation Function

The evaluation function is one of the most important functions in the solution method. It consists of 3 subfunctions. The evaluation function sums the penalty calculated by the subfunctions and puts this value in the first place in the solution array.

```plaintext
members[i][0]=(int)CalculateTiming(pol,porders,coc,edt,i);
members[i][0]+=(int)CalculateChangeovercosts(coc,i);
members[i][0]+=(int)CalculateLineCosts(lc,i);
```

Subfunction "CalculateTiming" calculates the timing of the different product on the lines. One input parameter here is the production capacity of each line. It is possible to manually specify an estimated time to finish a product order. An extra time is invoked when there is a changeover cost specified; this means that there is a time loss due to spooling. The actual penalty is looked up in a table called "extdeliverytimes".

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>penalty</td>
<td>21</td>
<td>18</td>
<td>15</td>
<td>12</td>
<td>9</td>
<td>6</td>
<td>3</td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 3.2: extdeliverytimes

Table 3.2 has to be interpreted as follows. The product order is asked for 7 hours from the beginning of the planning horizon so there is no penalty around this time. Earlier delivery is penalized with 3 cost units per hour too early; late delivery with 8 cost units per hour too late. This encourages a Just In Time production strategy.

Thus, to calculate the penalty for a certain solution, the solution array has to be decoded and production times of product orders have to be calculated. The penalty for every late or early delivery is summed up and finally returned to the evaluation function.

Subfunction "CalculateChangeovercosts" calculates the cost of changeover between successive product orders. If the successive product orders belong to a different product group and there is need for a spooling (presence of C in Table 2.1 on page 7) then a cost is generated. To be able to differ a successive product order of the different lines the solution array needs to be decoded. This cost is summed for the whole solution array and returned to the evaluation function.

Subfunction "CalculateLineCosts" represents the fact that some lines are better equipped to produce a certain product order, but they can be produced on another line provided an extra
effort. A cost is incurred if the product order is produced on a line that needs this extra effort. These costs are summed up for the whole solution array and returned to the evaluation function.

There is functionality in the model to track back the cause of the summed up cost. It is possible to see in the suggested solution which product order cause which kind of cost. This computable heavy functionality is turn of in the experiments.

The solution representation and the evaluation function currently do not allow to shut down production or product against stock of a bulk product. This possibly incurs costs by forcing product orders to be produced too early. This negative effect could be resolved by adding product orders with special constraints (no penalty for early or late production).

### 3.3 Genetic Algorithm (GA)

In addition to pure local improvement search, total enumeration, and branch-and-bound approaches there are meta heuristics (enhancing the local improvement search) search algorithms. This means that it is possible to find good solutions but we can not prove any optimality nor degree of optimality. [Rardin(2000)]. Three widespread techniques are Simulated Annealing, Tabu-Search, and Genetic Algorithms. Their use on scheduling problems are discussed by [Pinedo(1995)]

Genetic algorithms have been used for scheduling problems since the 1970s. Genetic algorithms have their fundamental base in evolutionary biology. Techniques used are inspired on inheritance, mutation, and natural selection. Solutions were originally represented by binary strings but there now exist many different representations. Many different approaches are possible to suit the practical problem.

What follows is a short description of a standard Genetic Algorithm. The algorithm starts by generating new members for the first generation. The fitness of the whole population is evaluated in each generation. This means that there must be a method to measure the fitness of a possible solution: the fitness function. The next generation is born by copying some of the members of the current generation with some advantage towards their fitness. Some other members are modified from the current generation. Two modifying operators are mutation and crossover. Mutation typically involves flipping one bit from its originally state in the binary presentation. In the used path representation one possibility is to exchange the place of two numbers randomly in the solution. The purpose of mutation is to avoid getting stuck in local optima. The crossover operator is explained in the next section. The GA approach used here is from the generational category. This means that the whole population is replaced in each generation, as opposed to steady-state mode where only one individual is replaced. The ”next generation” becomes the current in the next iteration. This whole process is iterated a number of times. A stopping criteria often used is the number of iterations. Another possible criteria is to take this decision after examination of the diversity of the population.

#### 3.3.1 Different Crossover Mechanisms

Many different crossover mechanisms exists. [Larranaga et al.(1999)] discuss in their review up to 16 different mechanisms for path based crossover. It has been claimed that an algorithm is not a genetic algorithm if the crossover mechanism is not present [Davis(1991)].

The crossover mechanisms discussed here are stochastic, which means that two same parents can yield many different children.

### Greedy Subtour Crossover (GSX)

This crossover function proved to be fast and successful in ”A Fast TSP Solver Using GA on JAVA”[Sengoku and Yoshihara(1993a)].
They proposed a new crossover operator to obtain the longest sequence of subtours possible from both parents and called it ’Greedy Subtour Crossover (GSX)’’. They proved by experiments that this crossover operator is more effectively avoiding local minima than simulated annealing methods [Sengoku and Yoshihara(1993b)]. The mechanism is explained in the next paragraph and Figure 3.1.

![Figure 3.1: Greedy Subtour Crossover](image)

We explain the algorithm by using letters of the alphabet that could be representations of product orders. The algorithm starts by choosing two parents out of the population. We choose randomly one starting letter. We take in turns now a letter of parent one or a letter of parent two starting from the initial letter. We walk through the left hand side letters of parent one, the right hand side of parent two. We stop taking letters from one parent if we bounce on an already used letter or the limits of the chromosome. After this, the rest of the letters are added in a random order to the child.

**Position Based Crossover (POS)**

[Persson(1996)] mentions that Position Based Crossover has been used earlier successfully for scheduling problems in the work of [Syswerda(1991)] The mechanism is explained in the next paragraph and Table 3.3.

<table>
<thead>
<tr>
<th>parent one</th>
<th>4</th>
<th>5</th>
<th>1</th>
<th>6</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parent two</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>o</td>
<td>o</td>
<td></td>
<td>o</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>child</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3.3: Position Based Crossover

The algorithm starts again with selecting two parents. Half the positions are randomly selected from parent one and straight away copied into the same position of the child. Now we run through the other parent and select the numbers not selected yet. These numbers are added into the empty positions of the child in the same relative order as in the parent.

**3.3.2 Specification**

What follows is a more down to earth specification of the Genetic Algorithm.

We generate a starting solution by randomly shuffling the numbers of the product orders. We put 100 array-solutions together in one matrix. This is our starting generation. Now all 100
solutions are evaluated and sorted along their fitness function. The next generation is formed by 4 mechanisms:

- We copy the 30 fittest of the current generation.
- We crossover two random members of the current generation and do this to obtain 40 new members.
- We mutate (pairwise change of two product orders) some random chosen members of the current generation to obtain 20 new members.
- We generate 10 new members.

After evaluation there is sorting again. This is iterated several times. Then the best found solution so far is reported. Figure 3.2 explains it visually.

Figure 3.2: Specification Genetic Algorithm

There is also possibility to have different runs in one execution. The code is attached in appendix.

3.4 Tabu Search

Fred Glover, generally seen as the attributor of Tabu Search, has several publications where the technique is successfully applied in operations research.

Performing local search with only allowing improving moves leads an algorithm to get stuck in a local optima. Tabu Search is one technique that deals with this problem by allowing non-improving moves. Allowing non-improving moves could cause an algorithm to cycle. Tabu
Search prevents short term cycling by keeping track of the last \( x \) number of performed moves. These moves are put in a “tabu list” with “Last In Last Out” strategy. Only moves not present in the tabu list are allowed. The solution is evaluated after every move and an “incumbent solution” keeps track of the so far best found solution. The algorithm stops after a number of iterations or some other stopping criteria. Figure 3.3 explains it visually.

![Flowchart of Tabu Search](image)

Figure 3.3: Tabu Search

### 3.4.1 Specification

The used Tabu Search implementation is built on algorithm 12D described in [Rardin(2000)].

The representation used for the Genetic Algorithm is also suitable for Tabu Search. The same evaluation function and parameters can be used, which is very convenient for benchmarking the algorithms. The length of the tabu is 70\% of a full solution array. Stop criteria is a number of iterations. The move implemented is a pairwise change of two randomly chosen product orders not present in the tabu. Only the first chosen product order is put in the tabu list. An incumbent solution keeps track of the best found solution so far. The code is attached in appendix.

### 3.5 Pairwise Change

Pairwise change is a very simple, intuitive, but computational hard local search heuristic. The specification is as follows. Every product order is pairwise exchanged with another. Then an evaluation of the solution is executed. The move is accepted if there is improvement. The method is computational hard because of the very frequent evaluation. One solution to diminish this drawback is to only pairwise change within one line in the solution array; and this for all three lines. A drawback is the reduced line balancing.
3.6 A Combined Approach: Genetic Algorithm and Pairwise Change

We look for other possibilities to further improve the approaches presented. It is sometimes possible to incorporate extra information extracted from the problem in the otherwise fairly blind local search approaches.

[Goldberg(1989)] suggested to cross Genetic Algorithm with a local search algorithm to obtain a better local search. Many different approaches are possible. Sometimes this approach is referred to as a hybridization.

Combinations as "tabu search and genetic algorithm" approaches have been tested shortly but very soon it was obvious that there was no noticeable improvement in the result; only an increase of computation time.

Another technique "seeding" entails the usage of known good solutions in the starting generation of the genetic algorithm. This did not work out very well.

The result of the genetic algorithm approach and the pairwise change approach individually are promising enough (see later) to justify their combination. The pairwise change algorithm is put inside the outer loop of the genetic algorithm.
Chapter 4

Computational Results

This chapter describes the computation results obtained using the solution techniques discussed in the previous chapter. We start with taking a look at the AMPL/CPLEX results. An introduction to the setting of the experiments is given before going deeper in their results. We end with discussing the results of a combined approach.

4.1 AMPL Results

In this section we present the results from AMPL. The main advantage of the AMPL modelling approach is the guaranteed optimality.

Main purpose of the AMPL test is to check whether the problem is solvable using standard software.

A case with nine product orders is solvable within half a minute on a recent computer (Intel(R) Pentium(R) 4 CPU 2.80GHz, 1.00 GB of RAM). The large scale problem takes in some cases very long time to obtain a solution.

The results given are based on the same product order input data as the results given in the next sections. The changeover costs were randomly generated.

It takes up to two hours to solve the problem. However, same model with same complexity takes only 70 seconds in a similar case, with relaxed time constraints. One reason could be that the branch and bound algorithm or some other algorithm in the CPLEX software package manage to find many branches that can be terminated early on in the search.

Observation: Computation time is very variable and dependent on the input data.

4.2 Input Data

As mentioned earlier, the different products produced in AAK belong to different product categories (cf. Table 2.1 on page 7). The 40 product orders are randomly assigned to a specific category.

The product orders specify the product type, quantities, and time of delivery. The data for the product orders is randomly generated to create a single scenario in the same margins as the real world case based on a sample of product orders of one day.

As discussed earlier in the solution representation, the production is not fully flexible as some product orders are usually produced on a particular line. The data representing this inflexibility is modelled as a penalty and is randomly generated in the model along a reasonable estimation (in less than 30 % of the cases there is a random line cost assigned on a specific line).

The capacity of the three lines is determined by data obtained from the company.
4.3 Setting

It is difficult to construct a fair comparison between the different algorithms because of their different computational overhead. For example tabu search and pairwise change are computed in a very small inner loop on one solution. Most of the computation is occupied evaluating the evaluation function. A genetic algorithm on the other hand has sorting, copying and other overhead functions in the most visited loop. However, the evaluation function comes out as the most computational demanding procedure for all tested algorithms. We choose to keep the number of evaluations as a constant. Although it is is not best practice to halt an algorithm in the middle of its operation, this seems to be the most fair approach. We choose to cut the algorithm on 5200 function evaluations. This number is chosen on grounds of a full execution of the genetic algorithm setting. The 100 evaluations of the first generation and 51 next generations with each 100 function evaluations totals to 5200.

We also mention that the initial population is randomly generated for the genetic algorithm and that the initial solution is randomly generated for the other heuristics.

We do 10 000 runs to obtain a representable, still computable result. We calculate the average value and the standard deviation (σ) to make an estimate of the range of variation in the reported best solutions. The standard deviation gives us an idea how far or close from the average value the other values lean; if we make the assumption that the data is "normally" distributed. We also provide the best found solution and the CPU time used.

We introduce the t-test. This statistical test compares the actual difference between two means in relation to the variation in the data.

Thus we compare the best solution and an average solution with standard deviation on an equal base of evaluation function visits.

We do note that some algorithm, given extra evaluation functions, do not improve anymore while others do.

One has to do a lot of experiments to record and understand all these different behaviors. It is almost impossible to try out all the different settings and combinations of the discussed algorithms. This is not the goal of this research. However, we first make some choices and assumptions in the next section to limit the number of combinations. Later we state all results in a table and make an appropriate decision for a combined approach.

The computer used has the following specification: Mobile Intel(R)4 CPU 3.06 GHz, 1.00 GB of RAM.

4.4 Genetic Algorithm: Greedy Subtour Crossover versus Position Based Crossover

The two different Crossover Mechanisms are tested in a typical genetic algorithm setting. There are 51 iterations (age of the GA algorithm) in one run. The other parameters (number of copied fittest, crossovers, mutators, new chromosomes) are the same as in the GA setting described earlier. The results are stated in Table 4.1 on the following page.

Observation: The standard deviation is in both cases relatively small; which means that the two algorithms have a stable outcome. The t-test states a value of 450; the difference between the groups is very unlikely to be a coincidence. The Position Based Crossover gives much better results than the Greedy Subtour Crossover. The big difference in best and average solution between the two crossover mechanisms means also that the GSX algorithm, although performing very well for the Traveling Salesman Problem (TSP), is not suitable for this scheduling problem. GSX is also not the best operator to use with regards to the used solution representation. The third and second line are more likely to have more product orders after the GSX operator; which can have a disturbing effect on the whole algorithm. The POS Crossover is deployed in the rest of the research.
4.5 Improving Local Search: Full versus Line Pairwise Change

This test is done with the same number of evaluations (5200) as the crossover test. The pairwise change is executed on one chromosome from the solution representation of the Genetic Algorithm. Results are stated in Table 4.2.

<table>
<thead>
<tr>
<th>Number of Runs</th>
<th>Full Pairwise Change</th>
<th>Line Pairwise Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>10000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solutions Evaluated per Run</th>
<th>Full Pairwise Change</th>
<th>Line Pairwise Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>5201</td>
<td>5201</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU Time (in ms) per Run</th>
<th>Full Pairwise Change</th>
<th>Line Pairwise Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.7547</td>
<td>25.6078</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Best Solution in All Runs</th>
<th>Full Pairwise Change</th>
<th>Line Pairwise Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>281</td>
<td>383</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Average Value of Best Solutions</th>
<th>Full Pairwise Change</th>
<th>Line Pairwise Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>455.6093</td>
<td>771.8573</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Standard Deviation</th>
<th>Full Pairwise Change</th>
<th>Line Pairwise Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>149.9177957</td>
<td>167.1342111</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t-Test</th>
<th>Full Pairwise Change</th>
<th>Line Pairwise Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>140.8552141</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Comparing Pairwise Change

Figure 4.1: Steps in Full and Line Pairwise Change

**Observation:** The simplified example in figure 4.1 demonstrates that the line pairwise change only has to do four pairwise changes in one execution against 35 for the full pairwise change. This means that, given an equal number of evaluations, the line pairwise change is executed more often. The total computation time is approximately the same which confirms the evaluation function as the core computational factor.

However the full pairwise change is performing much better in the experiment; that has a confident value of 140 in the t-test. One reason for the better performance is that the full pairwise change is also moving the sentinel values and thus searching for a line balancing. The full line pairwise change is performing better for this scheduling problem and is deployed in the rest of the research.
4.6 Tabu Search Approach

The Tabu Search setting for the experiment is the following: 5201 iterations of the algorithm on one solution. The result is stated in Table 4.3 on the next page.

**Observation:** Table 4.3 on the following page does not express very good results for the Tabu Search approach. There is not much improvement on the solution compared to the random solution generated case. One reason could be that tabu search looks inefficiently for good solutions because of the many possible pairwise changes with improvement in evaluation function. Another reason could be the perhaps too plain implementation of the tabu search algorithm.

4.7 A Combined Approach: Genetic Algorithm and Pairwise Change

The goal is to obtain the best and average solution of the pairwise change algorithm combined with the stable outcome of the genetic algorithm.

The pairwise change is placed inside the loop for every generation of the Genetic Algorithm. The main difficulty was to stick to the 5200 evaluation function. This is why we brought the 52 generations back to only three. A full pairwise change is executed on the fittest of each generation. The result is stated in Table 4.3 on the next page.

**Observation:** The standard deviation of the combination is very low, which means we have a stable output solution. Best and average result are better then all the other results. The computational time per execution is only slightly higher than for pairwise change. Thus we can speak of a good improvement, a worthy trade-off between computational time and better results.

4.8 Adding CPU-time to a Combined Approach

We also give an idea how good the solution could become; given more computational time and evaluation functions. We elevated the three generations of the combination approach to five and performed a pairwise change on the twenty fittest chromosomes of each generation. The results stated in Table 4.3 on the following page for “Adding CPU” are the best but computation time and number of evaluation functions have increased dramatically.

4.9 Computational Comparison

We add the results for the following random generation setting in Table 4.3: a random generation of 100 solutions, a sort operation, solution with lowest cost value is considered the solution of that specific run. We repeat this for 10 000 runs. We consider this setting as a reference point to benchmark against.

4.10 Cost of Inflexibility

We try to answer the following question in this section: ”How costly is dedicating a product to a certain line?”

We do this by setting up the following experiment. First the genetic algorithm is run with no costs at all for producing a product order on a specific line. Secondly, the same genetic algorithm is run with a very high line cost (400). We compare both results. The idea behind this setup is that the genetic algorithm is forced to take other penalties to avoid the high penalty for producing on a specific line. The “statistics option” in the implemented genetic algorithm allows to indicate the cases where the high cost path is taken in the final solution so we can subtract the high cost to compare both. We also reduce the number of runs to 100. The reason
for this is the computational very expensive bookkeeping of the statistics. We examine the t-test for evaluating the validness of the experiment. The result is stated in Table 4.4.

**Observation:** The t-test still has a value of 16, a good amount that ensures that the difference between the groups is very unlikely to be a coincidence. The results for the no line costs case are better on all fronts. This means that it is difficult to obey the rules of limiting the products to lines (otherwise the cost would be similar in both cases). The average cost for the case with line costs is up to 25% higher. We should however not be too fixed on this number because the costs should be more validated with the real world scheduling problem.
Chapter 5

Conclusions and Future Work

5.1 Conclusions

- The thesis introduced a mathematical model and identified, implemented and demonstrated useful optimization techniques for the complex scheduling problem.

- The model can be refined. The Operation Research cycle could be iterated a number of times with as input feedback from AarhusKarlshamn. Not all research questions are treated in the research, but a build-up towards them has been created.

- Some opportunities to lower costs; increase productivity have been suggested during the implementation and demonstration.

- Reasonably good results have been obtained in the final stage. The suggested combination is fast and can handle different types of costs. This is promising regarding the support of decomposition.

5.2 Future Work

Future work could possibly include:

- Add inventory levels to the model.

- Explore the validity and tractability of the presented model further and adapt it consequently.

- The mathematical model could become the core of an interactive Decision Support System or an Expert System.

- The above mentioned system themselves can be part of a greater supply chain management system (considering actors up and down stream), such as an enterprise resource planning (ERP) system. This system could control the whole chain and all complex interactions from raw material purchase to end-customer delivery; considering the ”greater picture”.

- Customize to better assist the planner; such as answering available-to-promise questions to customers.

- Research the applicability of a similar approach to other industries

- Add functionality to measure the performance with respect to usage of resources (inventory, machine utilization, etc.)
Bibliography


Appendix A

AMPL Model

param po=43;
set Productorder:= 3..po;
set Line=1..3;
param n=14;
set Sequence=1..n;

param d{Productorder};
param c{Productorder,Productorder};
param f{Productorder,Line};
param m=n*6;

var O{Productorder,Productorder,Line,Sequence} binary;
var X{Productorder,Line,0..n} binary;
var Y{Productorder,Line,0..n} >=0;

minimize costs:
sum{p1 in Productorder,p2 in Productorder,l in Line,s in Sequence}(O[p1,p2,l,s]*c[p1,p2]);

s.t. ConnectionTime(p in Productorder,l in Line,s in Sequence):
    Y[p,l,s] >= (f[p,l]*X[p,l,s]+sum{p1 in Productorder}(Y[p1,l,s-1])-m*(1-X[p,l,s]));

s.t. ConnectionTimeTwo(p in Productorder,l in Line,s in Sequence):
    Y[p,l,s] <=(m*X[p,l,s]);

s.t. Demand{p in Productorder}:
    sum{s in Sequence,l in Line}Y[p,l,s] <= d[p];

s.t. ConnectionChangeover{p1 in Productorder,p2 in Productorder,l in Line,s in Sequence}:
    0[p1,p2,l,s] >= (X[p1,l,s]+X[p2,l,s-1]-1);

s.t. Initialsequence{p in Productorder,l in Line}:
    Y[p,l,0]=0;

s.t. Production{p in Productorder}:
    sum{l in Line,s in Sequence}X[p,l,s] = 1;

s.t. Sequencing{s in Sequence,l in Line}:
    sum{p in Productorder}X[p,l,s] <= 1;
s.t. InitialProduction\{p in Productorder, l in Line\}:
    X[p, l, 0] = 1;

s.t. LogicalSequencing\{l in Line, s in Sequence\}:
    \sum\{p in Productorder\}X[p, l, s] <= \sum\{p in Productorder\}X[p, l, s-1];
Appendix B
Java Code

B.1 Scheduling.java

```java
import java.util.*;

class Scheduling {

    public static void main(String args[]) {
        long beginning = System.currentTimeMillis();
        System.out.println("----START OF THE COMBINED GENETIC ALGORITHM----");
        System.out.println();

        Data tempdata = new Data();
        int[][] tempporders = tempdata.returnProductorders();
        // this is the fittest member after a chosen number of runs
        int[] bestOverall = new int[tempporders.length];
        for (int i = 0; i < bestOverall.length; i++)
            bestOverall[i] = 100000;
        double average = 0;
        int runnumber = 10;
        double[] runvalue = new double[runnumber];
        double calcstanddeviation = 0;
        for (int run = 0; run < runnumber; run++) // here you can choose the number of runs
            {
                System.out.println("-----> THIS IS RUN NUMBER: "+run+" <------");
                Data localdata = new Data();
                int[][] porders = localdata.returnProductorders();
                localdata.generateChangeovercosts();
                int[][] coc = localdata.returnChangeovercosts();
                //
                // localdata.printChangeovercosts();
                //
                // localdata.printPolproductivity();
                // localdata.generateLineCosts();
                //
                // localdata.printLineCosts();
                int[][] lc = localdata.returnLineCosts();
                //
                // localdata.printLineCosts();
                double[][] pol = localdata.returnPolproductivity();
                localdata.calculateExtdeliverytimes();
                int[][] edt = localdata.returnExtdeliverytimes();
                //
                // localdata.printExtdeliverytimes();
            }

        System.out.println("\n----- THIS IS RUN NUMBER: "+run+" <------");
        Data localdata = new Data();
        int[][] porders = localdata.returnProductorders();
        localdata.generateChangeovercosts();
        int[][] coc = localdata.returnChangeovercosts();
        //
        // localdata.printChangeovercosts();
        //
        // localdata.printPolproductivity();
        // localdata.generateLineCosts();
        //
        // localdata.printLineCosts();
        int[][] lc = localdata.returnLineCosts();
        //
        // localdata.printLineCosts();
        double[][] pol = localdata.returnPolproductivity();
        localdata.calculateExtdeliverytimes();
        int[][] edt = localdata.returnExtdeliverytimes();
        //
        // localdata.printExtdeliverytimes();
    }
}
```
// the birth of the initial generation
Generation mainGeneration=new Generation(100,porders.length);
Generation.evalcounter=0;
mainGeneration.Shuffle(0,99);
mainGeneration.Calculate(pol, porders, coc,lc,edt);// calculate the costs
mainGeneration.Sort();// we sort the generation, fittest first!

//
// System.out.println("tabu search: ");
// mainGeneration.TabuSearch(0,0,pol,porders,coc,lc,edt);
//
// System.out.println("full pair wise change: ");
// for(int i=1;i<5;i++)
// {
//     if(Generation.evalcounter>5200)break;
//     mainGeneration.PairwiseChange(0,0,pol,porders,coc,lc,edt);
// }
// mainGeneration.Sort();// we sort the generation, fittest first!

int end=51;// here you can choose how old we let the generation come
for(int age=1; age<=end; age++)
{
    System.out.print(" *");
    Generation intermediateGeneration=new Generation(100,porders.length);
    // we keep the fittest of the previous generation
    intermediateGeneration.Keep(0,29,mainGeneration);
    // we replace a chosen part of the generation by crossovers
    intermediateGeneration.CrossoverTwo(30,69,porders,mainGeneration);
    // we replace a chosen part of the generation by mutators
    intermediateGeneration.Mutate(70,89,porders,mainGeneration);
    // some new members are joining the generation
    intermediateGeneration.Shuffle(90,99);
    // calculate the costs
    intermediateGeneration.Calculate(pol, porders, coc,lc,edt);
    //
    intermediateGeneration.FullPairwiseChange(0,0,pol,porders,coc,lc,edt);
    intermediateGeneration.Sort();// we sort the generation, fittest first!
    mainGeneration=intermediateGeneration;
}
System.out.println();
// mainGeneration.Print(0,3);

average+=mainGeneration.members[0][0];
runvalue[run]=mainGeneration.members[0][0];
// the fittest of each total run is saved in bestOverall
if(mainGeneration.members[0][0]<bestOverall[0])
{
    for (int i=0;i<bestOverall.length;i++)
        bestOverall[i]=mainGeneration.members[0][i];
}
//
mainGeneration.PrintStatistics(0);
average = average / runnumber;
System.out.println("\naverage value: "+average);
for (int i = 0; i < runvalue.length; i++)
{calcstanddeviation += ((runvalue[i]-average)*(runvalue[i]-average));}
calcstanddeviation = calcstanddeviation / runnumber;
calcstanddeviation = Math.sqrt(calcstanddeviation);
System.out.println("\nstandard deviation : "+calcstanddeviation);

System.out.println("the 'fittest' of this test-run is");
for (int i = 0; i < bestOverall.length; i++)
System.out.print(bestOverall[i] + "\t");
long end = System.currentTimeMillis();
System.out.println("\nprocessing took: "+(end-beginning) + " milliseconds");
System.out.println("\nevalcounter is: "+Generation.evalcounter);

B.2 Generation.java

import java.util.*;

class Generation
{
    int maxeval = 5200;
    public static long evalcounter = 0;
    //possibility to switch of the overhead operations of bookkeeping the different
    //causes of the total penalty
    boolean statisticsOn = false;
    //each row represents a member, the first element of a row contains the function
    //value, the i value is a stop value for the first line, then the second line starts,
    //the 2 value is a stop value for the second line, then the third line starts.
    int[][] members;
    int[][][] statistics;
    Generation(int a, int b)//constructor
    {
        members = new int[a][b];

        for (int i = 0; i < members.length; i++)//put initial values (0,1,2,...)
        {
            for (int j = 0; j < members[i].length; j++)
            {
                members[i][j] = j;
            }
        }
        //three dimensional matrix; one dimension for each possible cause of penalty
        statistics = new int[members.length][3][members[0].length];
        for (int i = 0; i < statistics.length; i++)
        {
            for (int j = 0; j < statistics[i].length; j++)
            {

```
for(int k=0; k < statistics[i][j].length; k++)
    statistics[i][j][k]=0;
}
}
// a function to shuffle all the values randomly
void Shuffle(int a, int b)
{
    Random rand = new Random();
    for (int i = a; i <= b; i++)
    {
        for (int j = 1; j < members[i].length; j++)
        {
            int randomPosition = rand.nextInt(members[i].length-1)+1;
            int temp = members[i][j];
            members[i][j] = members[i][randomPosition];
            members[i][randomPosition] = temp;
        }
    }
}
void Print(int a, int b) // print function
{
    for (int i = a; i <= b; i++)
    {
        for (int j = 0; j < members[i].length; j++)
        {
            if (j==0)
            {
                System.out.print("member: "+i+ " ---> value: "+members[i][0]);
                System.out.print("line 1: ");
            }
            else
            {
                switch(members[i][j])
                {
                    case 1: {System.out.print("line 2: "); break;}
                    case 2: {System.out.print("line 3: ");break;}
                    default: {System.out.print(members[i][j] + "\t");break;}
                }
            }
        }
        System.out.println();
    }
}
// this function calls for calculating the different penalties, adds them and puts
// this value in the first element of the considered row.
void Calculate(double[][] pol, int[][] porders, int[][] coc, int[][] lc, int[][] edt)
{
    if(statisticsOn)clearStatistics();
    for (int i = 0; i < members.length; i++)
    {
members[i][0]=0;
members[i][0]=(int)CalculateTiming(pol,porders,coc,edt,i);
members[i][0]+=(int)CalculateChangeovercosts(coc,i);
members[i][0]+=(int)CalculateLineCosts(lc,i);
}
}
//same as the previous function but with possibility to specify a member
int Calculate(double[][] pol,int[][] porders,int[][] coc,int[][] lc,int[][] edt, int now)
{
    if(statisticsOn)clearStatistics(now);
    members[now][0]=0;
    members[now][0]=(int)CalculateTiming(pol,porders,coc,edt,now);
    members[now][0]+=(int)CalculateChangeovercosts(coc,now);
    members[now][0]+=(int)CalculateLineCosts(lc,now);

    return members[now][0];
}
//this function calculates the timing of production on the different lines
//and penalizes early and late production.
double CalculateTiming(double[][] pol,int[][] porders,int[][] coc,int[][] lc,int[][] edt, int now)
{
    evalcounter++; 
    int temp=0;
    double[] timing=new double[members[0].length];
timing[0]=0;
    int integertiming=0;
    int selector=0;
    //selector equal to 0 means line 1, equal to 1 line 2, and equal to 2 line 3
    for (int i=1;i <members[0].length;i++)
    {
        if(members[now][i]==1 || members[now][i]==2)
        {
            if(members[now][i]==1){selector=1;}
            else {selector=2;}
            timing[i]=0;
        }
        else
        {
            timing[i]+=timing[i-1]+pol[members[now][i]][selector];
        }
        //an extra time is invoked when there appears a change over cost
        if(i!=members[0].length-1)
        {
            if(coc[members[now][i]][members[now][i+1]]!=0)
            {
                timing[i]+=0.1;
            }
        }
        else
        {
            timing[i]+=timeing[i-1]+pol[members[now][i]][selector];
        }
    }
    /* //case where only too late is penalized
    int penalty=30;
    if(timing[i]>porders[members[now][i]][1])
    {
        temp+=(penalty*(timing[i]-porders[members[now][i]][1]));
        if(statisticsOn)addToStatistics(now,0,members[now][i]);
    }
*/
/*
 //case where every product has a specific penalty for being produced in a timeslot
 integer timing=(integer)timing[i];
 if(integer timing>edt[0].length-1)integer timing=edt[0].length-1;
 temp+=edt[members[now][i][integer timing];
 if(statisticsOn)
   if(edt[now][integer timing]>0)
     addToStatistics(now,0,members[now][i]);
 }
 return temp;
}
//this function calculates the cost of changing between different products
double CalculateChangeovercosts(int[][] coc,int now)
{
  int temp=0;
  for (int i=2;i <members[0].length;i++)
  {
    temp+=coc[members[now][i-1]][members[now][i]];
    if(statisticsOn)
      if(coc[members[now][i-1]][members[now][i]] > 0)
        addToStatistics(now,1,members[now][i]);
  }
  return temp;
}
//this function calculates the cost of producing a product on a specific line
double CalculateLineCosts(int[][] lc,int now)
{
  int temp=0;
  double[] timing=new double[members[0].length];
  timing[0]=0;
  int selector=0;
  //selector equal to 0 means line 1, equal to 1 line 2, and equal to 2 line 3
  for (int i=1;i <members[0].length;i++)
  {
    if(members[now][i]==1 || members[now][i]==2)
    {
      if(members[now][i]==1){selector=1;}
      else {selector=2;}
      timing[i]=0;
    }
    else
    {
      temp+=lc[members[now][i]][selector];
      if(statisticsOn)
        if(lc[members[now][i]][selector]>0)
          addToStatistics(now,2,members[now][i]);
    }
  }
  return temp;
}
```java
// this function sorts the Generation; fittest members first!
void Sort()
{
    int min;
    for (int i = 0; i < members.length-1; i++)
    {
        min=i;
        for (int j = i+1; j < members.length; j++)
        {
            if(members[j][0]<members[min][0])min=j;
        }
        Swap(min,i);
    }
}

void Swap(int a,int b)//assists the Sort function
{
    int temporary;
    for (int j=0;j < members[a].length; j++)
    {
        temporary=members[a][j];
        members[a][j]=members[b][j];
        members[b][j]=temporary;
    }
}

// this function copies a specified number of members from one generation to another.
void Keep(int start,int end,Generation previous)
{
    for (int i = start; i <= end; i++)
    {
        for (int j = 0; j < members[0].length; j++)
        {
            members[i][j]=previous.members[i][j];
        }
    }
}

// Greedy Subtour Crossover
void CrossoverOne(int start,int end,int[][] porders,Generation previous)
{
    Random rand = new Random();
    int father;
    int mother;
    for(int now=start; now<=end;now++)
    {
        // an array to indicate which product orders are processed already
        int[] usage=new int[members[0].length];
        for(int i=0;i<usage.length;i++)usage[i]=0;

        father=rand.nextInt(members.length);
        do
        {
            mother=rand.nextInt(members.length);
        }
        while(father==mother);

        // Greedy Subtour Crossover...
    }
```
while (mother == father);

int r = rand.nextInt(members[0].length - 1) + 1; // choose a random value, not zero
int ra = 0, rb = 0;
for (int i = 1; i < members[0].length; i++)
{
    if (previous.members[father][i] == r) { ra = i - 1; } // find the random in A
    if (previous.members[mother][i] == r) { rb = i + 1; } // find the random in B
}

int[] achrom = new int[members[0].length];
for (int i = 0; i < achrom.length; i++) { achrom[i] = 0; }
int ap = 0;
int[] bchrom = new int[members[0].length]; //
for (int i = 0; i < bchrom.length; i++) { bchrom[i] = 0; }
int bp = 0;

int possib = 0;
while (ra > 0 || rb < members[0].length)
{
    if (ra > 0)
    {
        possib = previous.members[father][ra];
        ra--;
        if (usage[possib] == 0)
        {
            achrom[ap] = possib;
            usage[possib] = 1;
            ap++;
        }
        else { ra = 0; }
    }
    if (rb < members[0].length)
    {
        possib = previous.members[mother][rb];
        rb++;
        if (usage[possib] == 0)
        {
            bchrom[bp] = possib;
            usage[possib] = 1;
            bp++;
        }
        else { rb = members[0].length; }
    }
}
int len = 0;
for (int i = 1; i < usage.length; i++)
{ if (usage[i] == 0) len++; }

int[] help = new int[len];
int h = 0;
for(int i =1; i <usage.length;i++)
    {if(usage[i]==0) {help[h]=i;h++;}}

for (int i = 0; i < help.length; i++)
{
    int randomPosition = rand.nextInt(help.length);
    int temp = help[i];
    help[i] = help[randomPosition];
    help[randomPosition] = temp;
}

members[now][0]=0;
int index=0;
for(int i=achrom.length-1;i>=0;i--)
    {if(achrom[i]!=0){index++;members[now][index]=achrom[i];}}
index++;
members[now][index]=r;

for(int i=0;i<bchrom.length;i++)
    {if(bchrom[i]!=0){index++;members[now][index]=bchrom[i];}}

for(int i=0;i<help.length;i++)
    {index++;members[now][index]=help[i];}
}

//position based crossover
void CrossoverTwo(int start,int end,int[][] porders,Generation previous)
{
    Random rand = new Random();
    int father;
    int mother;
    //We crossover in the following way:
    for(int now=start; now<=end;now++)
    {
        //an array to indicate which p o are processed already
        int[] usage=new int[members[0].length];
        int[] help=new int[members[0].length];
        for(int i=0;i<usage.length;i++){usage[i]=0;help[i]=0;}

        father=rand.nextInt(members.length);
        do
            {mother=rand.nextInt(members.length);}
        while(mother==father);
        father=1;
        mother=2;
        for (int i=1; i<members[i].length;i++)
        {
            members[now][i]=0;
            double d = rand.nextDouble();
            if (d>0.5)
{ members[now][i]=members[father][i];
  usage[members[father][i]]=1;
}

int j=0;
for (int i=1; i<members[i].length;i++)
{
  if (usage[members[mother][i]]==0)
  {
    help[j]=members[mother][i];
    j++;
  }
}

j=0;
for (int i=1; i<members[i].length;i++)
{
  if (members[now][i]==0)
  {
    members[now][i]=help[j];
    j++;
  }
}

}

void Mutate(int start,int end,int[][] porders, Generation previous)
//We change pairwise two randomly chosen bits.
{
  Random rand = new Random();
  for (int counter = start; counter <= end; counter++)
  {
    int firstGene;
    int secondGene;
    int temp;
    int mutator=rand.nextInt(members.length);
    for (int x=0;x<members[counter].length;x++)
    {
      members[counter][x]=previous.members[mutator][x];
    }
    firstGene = rand.nextInt(members[counter].length-1)+1;
    do
    {
      secondGene = rand.nextInt(members[counter].length-1)+1;
      while(firstGene==secondGene);
      temp=members[counter][firstGene];
      members[counter][firstGene]=members[counter][secondGene];
      members[counter][secondGene]= temp;
    }
  }
  //We change pairwise two chosen bits within a line.
  //The change is accepted if there is an improvement.
  void PairwiseChange(int start,int end,double[][] pol,int[][] porders,int[][] coc,int[][] lc,
```csharp
int linechangeone=0;
int linechangetwo=0;
for (int counter = start; counter <= end; counter++)
{
    if(Generation.evalcounter>maxeval)break;
    for (int i=1;i <members[0].length;i++)
    {
        if(members[counter][i]==1)linechangeone=i;
        if(members[counter][i]==2)linechangetwo=i;
    }
    if(linechangetwo<linechangeone)
    {
        int temp=linechangeone;
        linechangeone=linechangetwo;
        linechangetwo=temp;
    }
    int before=0;
    int after=0;
    for (int i=1;i <linechangeone;i++)
    {
        if(Generation.evalcounter>maxeval)break;
        for (int j=i+1;j < linechangeone;j++)
        {
            if(Generation.evalcounter>maxeval)break;
            before=members[counter][0];
            swaplocal(i,j,counter);
            after=Calculate(pol,porders,coc,lc,edt,counter);
            if(Generation.evalcounter>maxeval)break;
            if(after > before)
            {
                swaplocal(i,j,counter);
                members[counter][0]=before;
                if(statisticsOn)
                    members[counter][0]=Calculate(pol,porders,coc,lc,edt,counter);
            }
        }
    }
    for (int i=linechangeone+1;i <linechangetwo;i++)
    {
        if(Generation.evalcounter>maxeval)break;
        for (int j=i+1;j < linechangetwo;j++)
        {
            if(Generation.evalcounter>maxeval)break;
            before=members[counter][0];
            swaplocal(i,j,counter);
            after=Calculate(pol,porders,coc,lc,edt,counter);
            if(after > before)
            {
                swaplocal(i,j,counter);
                members[counter][0]=before;
            }
        }
    }
}
```
if (statisticsOn)
    members[counter][0] = Calculate(pol, porders, coc, lc, edt, counter);
}

for (int i = linechangetwo + 1; i < members[0].length; i++)
{
    if (Generation.evalcounter > maxeval) break;
    for (int j = i + 1; j < members[0].length; j++)
    {
        if (Generation.evalcounter > maxeval) break;
        before = members[counter][0];
        swaplocal(i, j, counter);
        after = Calculate(pol, porders, coc, lc, edt, counter);
        if (after > before)
        {
            swaplocal(i, j, counter);
            members[counter][0] = before;
            if (statisticsOn)
                members[counter][0] = Calculate(pol, porders, coc, lc, edt, counter);
        }
    }
}

void swaplocal(int i, int j, int now) // assists PairwiseChange
{
    int temporary = members[now][i];
    members[now][i] = members[now][j];
    members[now][j] = temporary;
}

// We change pairwise two chosen bits over the whole array.
// The change is accepted if there is an improvement.
void FullPairwiseChange(int start, int end, double[][] pol, int[][] porders, int[] coc, int[] lc, int[] edt)
{
    for (int counter = start; counter <= end; counter++)
    {
        if (Generation.evalcounter > maxeval) break;
        int before = 0;
        int after = 0;
        for (int i = 1; i < members[0].length; i++)
        {
            if (Generation.evalcounter > maxeval) break;
            for (int j = i + 1; j < members[0].length; j++)
            {
                before = members[counter][0];
                swaplocal(i, j, counter);
                after = Calculate(pol, porders, coc, lc, edt, counter);
                if (Generation.evalcounter > maxeval) break;
                if (statisticsOn)
                    if (after > before)}
void TabuSearch(int start, int end, double[][] pol, int[][] porders, int[][] coc, int[][] lc, int[][] edt) {
    Random rand = new Random();
    int first;
    int second;
    int temp;

    for (int counter = start; counter <= end; counter++) {
        int[] incumbent = new int[members[counter].length];
        for (int i = 0; i < incumbent.length; i++) incumbent[i] = members[counter][i];
        int[] tabu = new int[(int)(members[counter].length * 0.7)];
        for (int i = 0; i < tabu.length; i++) tabu[i] = 0;
        int c = 0;
        int value;
        do {
            do first = rand.nextInt(members[counter].length - 1) + 1;
            while (inTabu(first, tabu));
            do second = rand.nextInt(members[counter].length - 1) + 1;
            while (first == second || inTabu(second, tabu));
            tabu = putInTabu(first, tabu);
            swaplocal(first, second, counter);
            value = Calculate(pol, porders, coc, lc, edt, counter);
            if (value < incumbent[0]) {
                for (int i = 0; i < incumbent.length; i++) incumbent[i] = members[counter][i];
                c++;
            }
        } while (c < 10000);
        //for (int i = 0; i < tabu.length; i++) System.out.print(" "+tabu[i]);
        //System.out.println();
        for (int i = 0; i < incumbent.length; i++) { members[counter][i] = incumbent[i]; }
    }
}

boolean inTabu(int first, int[] tabu) //assists TabuSearch
{
boolean temp=false;
for (int i=0; i < tabu.length; i++)
    if(tabu[i]==first)temp=true;
return temp;
}
int[] putInTabu(int first, int[] tabu) //assists TabuSearch
{
    int tempone=0;
    for (int i=0; i < tabu.length-1; i++) tabu[i]=tabu[i+1];
    tabu[tabu.length-1]=first;
    return tabu;
}
void clearStatistics() //initializes and clears the statistics
{
for (int i = 0; i < statistics.length; i++)
{
    for (int j = 0; j < statistics[i].length; j++)
    {
        for(int k=0; k < statistics[i][j].length; k++)
            statistics[i][j][k]=0;
    }
}
//initializes and clears a specified member of the statistics
void clearStatistics(int i)
{
    for (int j = 0; j < statistics[i].length; j++)
    {
        for(int k=0; k < statistics[i][j].length; k++)
            statistics[i][j][k]=0;
    }
}
//adds the specified cause to the statistics
void addToStatistics(int now, int measure, int add)
{
    for (int i=0; i < statistics[0][0].length; i++)
    if(statistics[0][measure][i]==0)
    {
        statistics[now][measure][i]=add;
        add=0;
        break;
    }
}
void PrintStatistics(int i) //prints the statistics
{
    System.out.println("\ntiming statistics:");
    for (int j = 0; j < statistics[0][0].length; j++)
    {
        System.out.println(statistics[i][0][j]+"\t");
    }
    System.out.println("\nchange over statistics:");
for (int j = 0; j < statistics[0][0].length; j++)
{
    System.out.print(statistics[i][1][j]+"\t");
}
System.out.println("line statistics: ");
for (int j = 0; j < statistics[0][0].length; j++)
{
    System.out.print(statistics[i][2][j]+"\t");
}

B.3 Data.java

import java.util.*;

class Data {
    private int[][] changeovercosts;
    private double[][] polproductivity;
    private int[][] linecosts;
    private int lines=3;
    private int[][] extdeliverytimes;
    private int timehorizon=12;
    //matrix for productorders: {order number,time of delivery(1-12), quantity(1-30)}
    private int[][] productorders=
    {
        {0,0,0}, //for normalisation.
        {1,0,0}, //a sentinel indicating the end of the first line, start of the second
        {2,0,0}, //a sentinel indicating the end of the second line, start of the third
        {3,5,14},
        {4,7,3},
        {5,9,18},
        {6,8,30},
        {7,7,9},
        {8,8,5},
        {9,5,23},
        {10,4,23},
        {11,11,14},
        {12,6,24},
        {13,5,6},
        {14,3,10},
        {15,10,16},
        {16,5,3},
        {17,4,5},
        {18,11,24},
        {19,5,25},
        {20,4,27},
        {21,6,23},
        {22,12,10},
        {23,6,26},
        {24,12,1},
        }
private String[][] changeovermatrix =
{
};

int[][] returnProductorders()
{
    int[][] temp = new int[productorders.length][productorders[0].length];
    for (int i = 0; i < productorders.length; i++)
    {
        for (int j = 0; j < productorders[i].length; j++)
        {
            temp[i][j] = productorders[i][j];
        }
    }
    return temp;
}

void generateChangeovercosts()
{
    changeovercosts = new int[productorders.length][productorders.length];
    Random rand = new Random();
    int[] category = new int[productorders.length];
    // randomly assign the product to a product group
    for (int i = 0; i < productorders.length; i++)
category[i]=rand.nextInt(changeovermatrix.length-1)+1;
for (int i = 0; i < changeovercosts.length; i++)
{
  for (int j = 0; j < changeovercosts[i].length; j++)
  {
    if(i==j || i==0 || i==1 || i==2 || j==0 || j==1 || j==2 )
    {
      changeovercosts[i][j]=0;
    } else
    {
      if (changeovermatrix[category[i]][category[j]]==”C”)
        changeovercosts[i][j]=15;
      else {changeovercosts[i][j]=0;}
    }
  }
}
void printChangeovercosts()
{
  System.out.println();
  for (int i = 0; i < changeovercosts.length; i++)
  {
    for (int j = 0; j < changeovercosts[i].length; j++)
    {
      System.out.print(changeovercosts[i][j] +”	”); if(j==(changeovercosts[0].length-1))System.out.println();
    }
    System.out.println();
  }
}
int[][] returnChangeovercosts()
{
  return changeovercosts;
}
void generatePolproductivity()
{
  polproductivity=new double[productorders.length][lines];
  for (int i = 0; i < polproductivity.length; i++)
  {
    for (int j = 0; j < polproductivity[i].length; j++)
    {
      if(j!=2)
        polproductivity[i][j]=productorders[i][2]/21.0;
      else polproductivity[i][j]=productorders[i][2]/30.0;
    }
  }
}
void printPolproductivity()
{
  System.out.println();
  for (int i = 0; i < polproductivity.length; i++)
  {
for (int j = 0; j < polproductivity[i].length; j++)
{
    System.out.print(polproductivity[i][j]+"\t");
    if(j==(polproductivity[0].length-1))System.out.println();
}
}
double[][] returnPolproductivity()
{
    return polproductivity;
}
void generateLineCosts()
{
    linecosts=new int[productorders.length][lines];
    Random rand = new Random();
    for (int i = 0; i < linecosts.length; i++)
    {
        for (int j = 0; j < linecosts[i].length; j++)
        {
            double d = rand.nextDouble();
            if (d<0.3){linecosts[i][j]=rand.nextInt(30);}
            else {linecosts[i][j]=0;}

        }
    }
}
void printLineCosts()
{
    System.out.println();
    for (int i = 0; i < linecosts.length; i++)
    {
        for (int j = 0; j < linecosts[i].length; j++)
        {
            System.out.print(linecosts[i][j]+"\t");
            if(j==(linecosts[0].length-1))System.out.println();

        }
    }
}
int[][] returnLineCosts()
{
    return linecosts;
}
void calculateExtdeliverytimes()
{
    extdeliverytimes=new int[productorders.length][3*timehorizon+1];
    for (int i = 0; i < extdeliverytimes[0].length; i++)
    {
        extdeliverytimes[0][i]=0;
        extdeliverytimes[1][i]=0;
        extdeliverytimes[2][i]=0;
    }
}
int perfecttime;
for (int i = 3; i < extdeliverytimes.length; i++)
{
    perfecttime=productorders[i][1];

    for (int j = perfecttime-1; j >= 0; j--)
    {
        extdeliverytimes[i][j]=3*(perfecttime-j);
    }
    extdeliverytimes[i][perfecttime]=0;
    for (int j = perfecttime+1; j < extdeliverytimes[i].length; j++)
    {
        extdeliverytimes[i][j]=8*(j-perfecttime);
    }
}
}
void printExtdeliverytimes()
{
    System.out.println("\nExtension delivery times: ");
    for (int i = 0; i < extdeliverytimes.length; i++)
    {
        System.out.println();
        for (int j = 0; j < extdeliverytimes[i].length; j++)
        {
            System.out.print(extdeliverytimes[i][j]+"\t");
            if(j==(extdeliverytimes[0].length-1))System.out.println();
        }
    }
}
int[][] returnExtdeliverytimes()
{
    return extdeliverytimes;
}