TUCS

Mika Hirvikorpi

On the Tactical Level Production Planning in Flexible Manufacturing Systems

Turku Centre *for* Computer Science

TUCS Dissertations
No 66, November 2005

On the tactical level production planning in flexible manufacturing systems

Mika Hirvikorpi

To be presented, with the permission of the Faculty of Natural Sciences and Mathematics for public criticism in the Auditorium of PharmaCity in November 25th, 2005 at 12:00.

Turku University
Department of Information Technology
Lemminkäisenkatu 14 A, 20520 Turku
2005

Supervised by

Professor Olli Nevalainen
Department of Information Technology
Turku University
Turku, Finland

Professor Timo Knuutila
Department of Information Technology
Turku University
Turku, Finland

Reviewed by

Professor Kauko Leiviskä
Department of Process and Environmental Engineering
University of Oulu
Oulu, Finland

Professor Gürsel A. Süer
Department of Industrial and Systems Engineering
Ohio University
Athens, Ohio, United States of America

Opponent

Professor Jyrki Nummenmaa
Department of Computer science
University of Tampere
Tampere, Finland

ISBN 951-29-2789-6

ISSN 1239-1883

Painosalama Oy, 2005

Turku, Finland

Acknowledgements

It is a moment of great joy to finish my studies and earn one of the highest degrees of education. Learning and understanding are to me one of the most important things in life. I wish to thank Turku University and Turku Centre for Computer Science for expanding my understanding on many fields of science. I also want to thank them for their financial support and an inspiring environment to do research.

I would like to thank my supervising professors Olli Nevalainen and Timo Knuutila who gave me an opportunity to continue my studies after completing my master of science degree. They have also offered me their knowledge and experience in all phases of my work.

Next, I would like to thank the reviewers of my work who helped to finalize it by providing their analysis of its' content. They also gave something to think about when continuing my research.

The encouragement from my parents has been a significant factor in completing my work. They have always believed that good education is an important factor in successful life.

Finally, I would like to thank the most important person in my life, my fiancée Terhi, she gave me back my inspiration and motivation to science. Without her support and love this dissertation might have not finished.

Turku, September 5, 2005

Mika Hirvikorpi

Preface

This work concentrates mainly on one-machine optimization problems in the field of printed circuit board assembly. The modelled problems can be applied also on several other applications of the flexible manufacturing systems.

The introduction explains the concept of production planning and describes the different levels of planning. Some historical background of the manufacturing industry is also given. An overall description of the flexible manufacturing systems and the problems related to their optimization is given next. Finally, printed circuit board assembly environment and the working principles of the assembly machines as well as a classification of the PCB assembly problems is given.

Next, the different methods used in solving the combinatorial optimization problems (COP) in the publications of this work are presented. Three aspects for solving COPs are considered: heuristics, lower and upper bounding and exact methods.

The publications of this work are summarized in section 3. For each publication we give a general description of the considered problem, ideas about the solution methods and the conclusions based on the empirical findings of these studies.

Section 4 summarizes the results of this work in a general manner. Section 5 lists all the references used in the introduction and Section 6 consists of the original publications.

Contents

List of acronyms

CNC	computer numerical control
CO	combinatorial optimization
COP	combinatorial optimization problem
FMS	flexible manufacturing system
GA	genetic algorithm
IMS	intelligent manufacturing system
JGP	job grouping problem
KTNS	keep tool needed soonest
NP	nondeterministically polynomial problems
PCB	printed circuit board
SOP	stochastic optimization problem
SPT	shortest processing time
TS	tabu search
TSP	tool switch policy

List of original publications

I. Knuutila T., Hirvikorpi M., Johnsson M. and Nevalainen O.S., Grouping PCB assembly jobs with typed component feeder units. *International Journal of Flexible Manufacturing*, 16(2), 2004.

II. Knuutila T., Hirvikorpi M., Johnsson M. and Nevalainen O.S., Grouping of PCB Assembly Jobs in the Case of Flexible Feeder Units. *Engineering Optimization*, 37(1), 2005.

III. Hirvikorpi M., Salonen K., Knuutila T. and Nevalainen O.S., The general two level storage management problem: a reconsideration of the KTNS-rule. To appear in *European journal of operational research*.

IV. Hirvikorpi M., Johnsson M., Knuutila T. and Nevalainen O.S., A General Approach to Grouping of PCB Assembly Jobs. To appear in *International Journal of Computer Integrated Manufacturing*, 2004.

V. Hirvikorpi M., Knuutila T. and Nevalainen O., Job ordering and management of wearing tools in flexible manufacturing. To appear in *Engineering Optimization*, 2004.

VI. Hirvikorpi M., Knuutila T., Leipälä T. and Nevalainen O.S., Job Scheduling and Management of Wearing Tools with Stochastic Lifetimes, Submitted for publication, 2004.

Chapter 1

Introduction

Industrialization was a significant step forward in producing wide variety of customer products in the nineteenth century. Before industrialism, the production volumes even for simple products like nails were quite small and the production process was very labour intensive. Now, more than two centuries later the industrialization still continues as manufacturing of complex products is automatized. The latest phase in this development is the rise of the intelligent manufacturing systems (IMS). Intelligence in this context usually refers to computers that guide the production and are able to make simple decisions by themselves. The complexity of IMSes is remarkable and their prices are very high, thus it is important to use them at optimal efficiency.

Optimizing the production of an entire production plant is not possible in practice because their mathematical models include thousands of variables and there are tens of ways to measure the efficiency of the production. Production plants typically have several concurrently operating production lines which are used for manufacturing multiple products or a single product. The objective in production planning is to create a schedule for the individual production steps which uses the production lines as efficiently as possible. The efficiency can either be measured by the cost, time or quality of the process. It is also possible to compromise between these and many other objectives in which case one faces a multiobjective optimization task.

From optimization perspective the factory (production plant) can be seen as a hierarchical system. On the highest level one must decide how the products are distributed to the production lines. It is possible that some products can be manufactured only on certain lines which set additional difficulties on the planning. Once the products have been distributed among the lines, the production on them is optimized separately. One must decide on the level of a single line, for example, the order and speed of manufacturing. Additional constraints, like due dates, of the products increase the complexity of the problem in practice. On the lowest level of production planning, decisions regarding how to use the individual machines to manufacture a single product or part of the product must be made.

The hierarchical way of looking the optimization problems of a factory is in many cases the only possible due to the complexity of the whole production plant. The approach does not guarantee the optimality of the production schedule, since the different levels of hierarchy do not interact. An optimal solution can only be guaranteed by defining the optimizing task as a single problem. It has, however, been demonstrated that in several practical cases the production efficiency can be improved by identifying so called bottleneck parts of the process. In flexible manufacturing systems, what this work is about, the bottleneck of the process is often a single CNC (computer numerical control) machine. Optimizing the production on the level of a single CNC machine often brings significant improvement in the total efficiency of the production [4,5,6,7].

In the following two sections we first discuss a specific area of flexible manufacturing systems, namely PCB (printed circuit board) assembly and then FMSes in general. Although PCB assembly lines are FMSes, we deal with them separately from the other flexible manufacturing systems. This separation is an artificial one, but in PCB assembly tool wear is not an important issue. On the other hand it is central in many other fields of FMSes.

1.1 Flexible manufacturing systems

The concept of a flexible manufacturing system (FMS) was introduced in the 60's in England. The goal was to automate the manufacturing processes of certain consumer products so that the production line would operate 24 hours a day and seven days a week without interruptions. This goal still remains a fantasy, but FMSes provide several other benefits, including [11]:

- homogenous and improved product quality;
- reduced production time;
- better utilization of human workers; and
- capability of producing several products with the same equipment.

The homogenous quality comes from the machines' capability to repeat the same operations with extreme precision which also improves the quality. In comparison to a human worker this is a clear advantage since it reduces the possibility of mistakes in the production. Machines are also faster in doing things which do not require reasoning. Instead of reserving a large number of people to perform a single task, FMS needs only a few people to operate it. The capability of manufacturing several products on a single line is possible because the production is usually guided by a computer numerical control (CNC). In CNC manufacturing, switching from the production of one product to another requires ideally only changing the control programs of the machines on the production lines.

1.1.1 A Typical flexible manufacturing system

A typical FMS consists of computer numerical controlled (CNC) workstations and material handling systems. The CNC workstations perform traditional operations of machining like drilling and welding. The material handling system interconnects the devices and may perform quality inspections for the parts to be manufactured, but it does not perform actual machining operations to the parts. The layout of an FMS gives an overall idea of how the system works and how the parts move in it. Commonly used layouts in FMSes are [12]:

- progressive layout;
- closed loop layout;
- ladder layout; and
- open field layout.

In the progressive layout, the part moves always forward on the production line. This is the simplest form of an FMS and layout. The closed loop layout is a bit more complicated since the part can skip machines on the production line. For example, if some machine is currently occupied then the part moves to some other machine and returns to this one later. It is also possible to use this type of FMS to manufacture different types of parts by skipping some of the machines completely. In ladder layout the part can go to any machine in any order because the machines are all connected to the same material transport line. The fact that the parts use same transport line limits the processing order and part movements in practice. The open field layout is the most complex one, since it is able to move the material in any order to any machine. There are usually support stations which control the movements of the parts. These support stations can also perform inspections and change tools to the machines. Figure 1 and Figure 2 show an example of each layout type.

The material handling system which is an integral part of an FMS can consist of many types of devices. Conveyor belts are the most common ones since they transport the parts from a machine to another. Automated guided vehicles (AGV) or carts are used to transport parts on longer distances. Some FMSes also include an automatic storage and retrieval system which transports finished products to storage and retrieves materials for the machines.

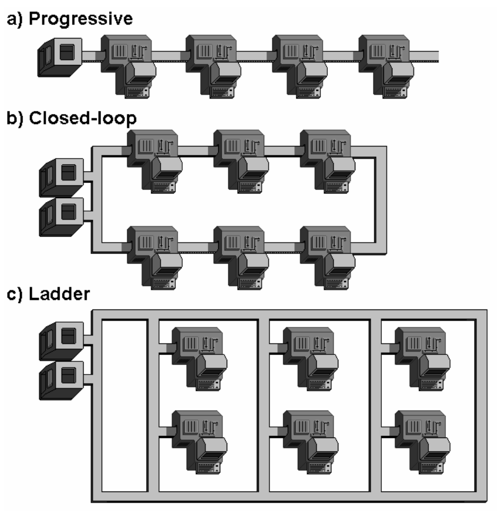


Figure 1. Examples of commonly used layouts in FMSes. The loading/ unloading stations are on the left of the figures.

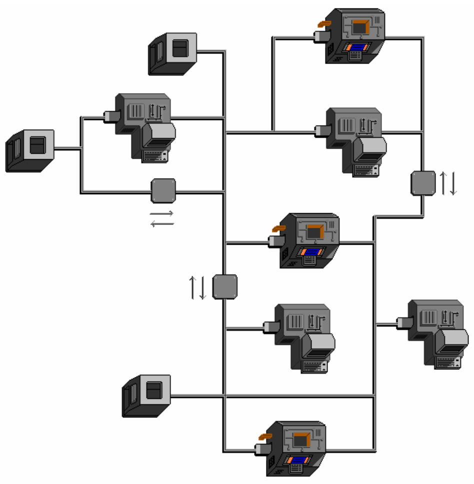


Figure 2. An example of an open-field layout of FMS. There are three loading/unloading stations in this particular FMS as well as three material transporting carts

1.1.2 Computer numerically controlled machines

The numerical control (NC) machines were developed shortly after World War II. These were the predecessors of the CNC machines. In NC machines, the instructions were fed in the form of punctuation cards or paper tapes. At the beginning of the 1960’s the first CNC machines were introduced by the MIT Servomechanisms laboratory. As comparison to the NC machines the programs were fed directly from a computer. This makes the control programs easier to update and store. The amount of data computers are able to store is also significantly greater than that of punctuation cards or paper tapes. The latest development of CNC machines is the possibility to feed data directly from CAD applications [13].

CNC machines radically extended the possibilities of automated manufacturing since they were able to perform many operations with greater accuracy than a human operator. The CNC machines can, for example, cut materials along curves as easily as along straight lines. They are also able to manufacture complex three-dimensional objects directly from CAD models.

An input of a CNC machine is a computer program and a block of material. The sophistication of the machine dictates what kind of programs it understands. The input can, for example, be a three-dimensional CAD model which is then translated into a series of simple commands. The block of material can be almost any kind of metal, wood, plastic or even stone depending on the type of the CNC machine.

CNC machines use tools to process the material blocks. Older CNC machines usually have only one type of tool which they use to process the incoming material blocks. More sophisticated machines have a tool magazine from which they retrieve the tools according to a control program. Since the operations are performed with changeable tools, one machine is able to perform various types of operations to the material blocks. A single machine can have one or several of the following common CNC tools:

- drill;
- lathe;
- mill;
- cutter; and
- weld.

All of these tools are used to shape the incoming material or to join pieces of material together. The techniques used in these tools vary greatly. Cutter, for example, can be a laser cutter or an EDM (electrical discharge machine). Figure 3 shows an example of a CNC machine.

Figure 3. An example of a CNC machining center [14]. This machine is capable of performing several traditional machining operations like milling, drilling and lathing.

1.1.3 Optimization problems in flexible manufacturing systems

An FMS may consist of a single CNC machine and a transport line or it can form a whole factory with a complex material handling systems and several CNC machines. Therefore, the number of problems related to FMSes is very large. The problems can be categorized to pre-production and production problems. The pre-production problems include the placement of the machines [15,16,17,18], the number and type of machines included in FMS and tool and material acquisition. In the following we recall the most studied production problems related to FMSes. The pre-production problems are out of the scope of this work.

The production problems consider the various aspects of optimizing the production when a set of FMSes (for example several production lines) and a set of jobs to be processed are given. The jobs bring additional constraints like due dates, tool requirements, order requirements, etc. make optimization more complicated.

On the highest level of optimization one has to decide, which line each product is manufactured on. In balancing problems the products must be distributed to the lines so that optimal throughput is reached. This can be done to either one product or multiple products. In the latter case we are talking about scheduling problems. In single production line problems we are given a set of products to be manufactured with an FMS. The main problem on this level is how to choose a proper manufacturing order of the products and how to allocate the resources, like tools, to the machines.

The most studied problems are, however, the single machine optimization problems. The tool switching (TS) problem is one of the most famous problems in the field. In the TS-problem one is given a set of products (jobs) to be processed on a single machine. This machine uses tools to process the jobs and each job may require one or several tools in the processing. The number of tools the machine can keep simultaneously in its tool magazine is limited. The objective is then to choose the processing order of the jobs and the tool management decisions so that the number of tool switches is minimal. This problem is known to be NP-hard [19] and therefore supposed to be difficult to solve exactly. The TS-problem has been studied by several authors including [19,20,21,22]. All these papers assume that the tools consume a uniform amount of space from the tool magazine. In many cases this assumption is too restrictive and therefore some authors have relaxed this assumption [23,24].

Making the tool management decisions is an important part of the tool switching -problem when the order of processing the jobs is fixed. For uniform sized tools this problem can be solved optimally with the KTNS (keep tool needed soonest) -rule [25]. We have studied this problem with tools of variable sizes in [24].

In the TS-problem one tries to minimize the number of tool switches because tool switches are considered to be so time taking that they consume most of the production time. Another possible objective is to minimize the average completion time of the jobs. This problem can be solved to optimality with the SPT (shortest processing time) -rule if we omit the tool switching related issues [26]. This problem can be generalized by assuming finite life-times for the tools. The problem of wearing tools has been studied [27] in the case of a single tool. We have extended this problem formulation in [28] to include the use of several tools and a tool magazine of limited capacity. We also study the stochastic version of the wearing tools problem [29] in which one assumes that tool life-times follow some probability distributions.

As discussed before, the material handling system is a central part of an FMS. Despite of this fact its modelling has been omitted in most studies by simply assuming that material movements require no time. The importance of modelling the material handling systems has, however, been pointed out by some authors, including [30,31,32]. Crama [33] introduces a general multiserver scheduling model to model the problem related to material handling system. This model consists of an input station, an output station, a set of machines and a set of servers. The manufactured products are initially in the output station from which they end up through the machines and servers to the output station. The model assumes that each machine is able to process one product at a time and that servers are used to load (unload) the products to (from) the machines. Further, each server can handle only one product at a time. The objective is to devise a schedule for the products and servers in an attempt to optimize the production from some perspective, like cycle time.

The flow shop scheduling problems model the material handling in FMSes and fit to the general model Crama [33]. In flow shop -problems the machines are in a form of a line or a circle and each product must be processed on each machine in the order in which they appear on the line. These problems can further be categorized to robotic, hoist and parallel flow shop –problems. In robotic flow shop –problems [34-39] the objective is to minimize the long-run cycle time with an assumption that the production continues infinitely long. Another assumption is the presence of a unique server only.

The hoist scheduling –problems [40-43] are closely related to the robotic flow shop –problems but they allow multiple servers. The products are also assumed to be identical in the hoist setting. Another major difference is that the robotic flow shop –problems assume a fixed processing time associated to each (product, machine) –pair whereas the hoist problems assume a time window. This means that the processing can last a variable time. This situation is common on complex PCB assembly lines.

The third category of flow shop problems, namely multiserver scheduling problems with parallel machines, assumes identical machines. It is also assumed that each product is processed only on a single machine. This approach has been studied for example in [44-47].

1.2 Printed circuit board assembly

Virtually every electronical device contains a printed circuit board (PCB) or several of them and therefore the number of PCBs produced worldwide every year is tremendous and increasing rapidly. Despite of the many advances in PCB production there remains still much to be improved in the usage level of the PCB production lines. In some cases the introduction of more complex machines has actually lowered the usage level of the equipment while increasing the efficiency in other respects, for example some machines are easier to operate and others can assemble more complex PCBs. A good introduction to PCB assembly in general is given in [8].

1.2.1 A typical PCB assembly line

Figure 4 contains a typical configuration of a PCB assembly line. The line begins with a loader from which the bare PCBs are entered to the line. The next device, which is either solder paste or glue dispenser, places solder or glue to the component connector positions. The solder is spread over a so called stencil which has holes on the component connections while glue is placed by a machine to the connector positions directly.

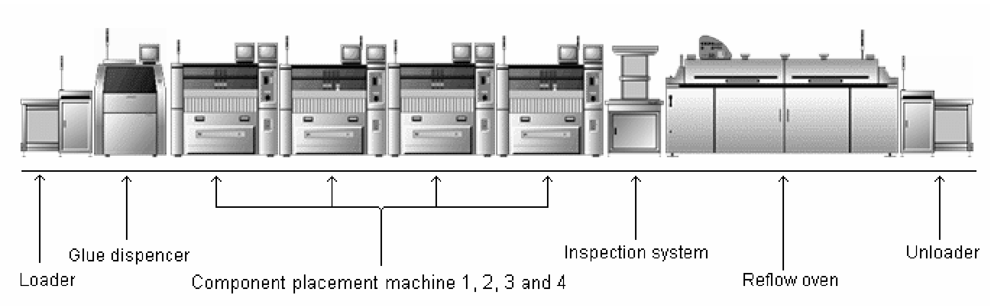


Figure 4. An example of PCB assembly production line [9].

After the soldering process there are one or several high-speed component placement machines and usually at least one high-precision machine. In the example line of Figure 4 the machines are identical. The usage of several machines on a line increases the throughput. It is also common that one machine is not able to place some component types at all and thus several different types of machines are needed. Once the components have been placed there is a visual inspection system which examines that the components are in their correct places and orientations. This is necessary because the high speed placement machines move the PCBs quite rapidly and the acceleration of the board can displace some components during the placement process.

The next phase of the process is the fixation of the components. This is done in a reflow oven which heats the PCBs and makes the solder paste or glue to melt and this way attaches the components to the PCB. Finally, the finished PCBs are unloaded from the production line.

1.2.2 Component placement machines

The bottleneck of the assembly process can be one of the component placement machines and therefore it is common to optimize the production from their perspective. The technical details of these machines vary greatly between machine manufacturers and models. One can, however, categorize them when omitting the technical details and concentrating on the main working principles. We mention three different types of placement machines here. Two of these are commonly used today but the axial machines are mentioned as a historical curiosity, only.

These placement machines have many common parts. First one of these is the PCB table which holds the bare PCB on which the components are placed. The placement is done by a printing head which receives the components from a feeder. In the printing head there are one or several nozzles which hold the component(s) when moving to the correct position above the PCB. Finally, the printing is always guided by a computer program which feeds commands to the machine. These commands are simple, for example they can order the printing head to move certain amount of centimetres on x-axis or y-axis.

The working principle of axial machines is shown in Figure 5. The placement of a single component begins by moving the table on which the PCB is to the right position. After this the printing head takes the next component from the feeder and places it to the PCB. This is iterated as long as all components have been placed. The axial machines are one of the oldest technologies used in component placement. They are quite inflexible because the component must be placed in the order placement to the component feeder. The capability of axial machines to place different types of components is also somewhat limited because they use the same printing head configuration for all components.

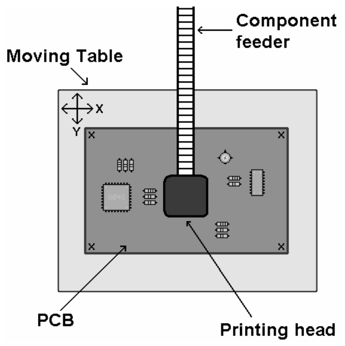


Figure 6. Axial placement machine.

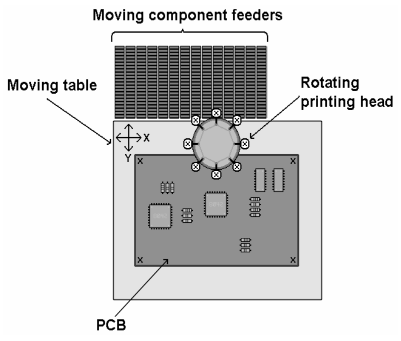


Figure 5. Rotary turret machine.

Rotary turret machines are more flexible than the traditional axial machines. They operate also on a moving table, but the component feeder and the printing head are more advanced. The placement in a rotary turret machine begins with component retrieval from the feeders. In the first step the feeders move to a position which enables the printing head to retrieve the component from them. The nozzles which are around the circle-shaped printing head hold the components during the placement. At the same time when a component is picked up from the feeder, some other component is placed on the opposite side of the printing head. After the placement and retrieval the head advances one nozzle position forward and performs the same operations. The operating principle of a rotary turret machine is illustrated in Figure 6.

Currently, the most advanced machine type is the pick-and-place machine, where the printing head moves and the table and feeders remain at fixed positions, see Figure 7. The placement operation begins by component retrieval from the feeders. The printing head moves to the feeder position in which the required component is. Next, the head picks the component up to its nozzle and moves to the correct placement position. Finally, it places the component to the desired position. There can be one or several nozzles in the printing head depending on the level of sophistication of the machine and thus it may be possible to pick up several components during one retrieval.

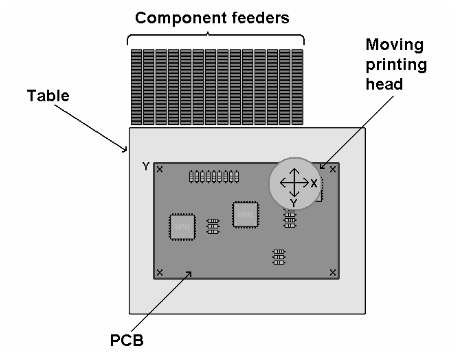


Figure 7. Pick-and-place machine.

The component feeders in turret and pick-and-place machines can hold thousands of components of several different types. The component feeders are side by side and each feeder contains several components of certain type. Figure 6 and Figure 7 demonstrate the so called tape feeders, but there are several other techniques including tray, track and bulk feeders. It is possible that the machine uses many different types of feeders and, in the most advanced machines, the feeder configuration can even be modified. See publications I-III of this thesis for more information on these variations.

1.2.3 Optimization problems in PCB assembly

The optimization problems in PCB assembly have been traditionally categorized according to the number of machines and the number of PCB types manufactured [10]:

- one PCB type and one machine (1-1);
- one PCB type and several machines (1-N);
- multiple PCB types and one machine (1-M); and
- multiple PCB types and machines (M-N).

The first category (1-1) of problems is further divided to four classes of problems in various studies.
In the feeder arrangement problems one attempts to organize the component types to the feeder slots in a way which maximizes the component placement speed. In placement sequencing problems one tries to minimize the component placement time by performing the placement in an order which minimizes the distance travelled by the placement head. In nozzle assignment problems one considers the various aspects of how to choose the multiple nozzles for the placement heads. In component allocation problems one attempts to organize the component

types to the feeder in a case where multiple copies of one component type may be placed to the feeder. The (1-N)-category consists of line balancing problems. The goal is to distribute the PCBs to different machines so that the workload is as balanced as possible.

The PCB assembly problems we study in this work belong to the (1-M)-category. In this category there are the problems concerning different setup strategies for single machine optimization. There are four major different setup strategies [10]:

- unique setup strategy;
- minimum setup strategy;
- group setup strategy; and
- partial setup strategy.

The first setup method considers the PCB types in isolation and attempts to minimize the placement time on the level of a single PCB type. In the minimum setup strategy the objective is to minimize the total component setup time. This is achieved by ordering the PCB types properly. In the group setup strategy the goal is to group the PCB types so that the number of setup occasions is minimized. The grouping is formed so that each group of PCB types within the grouping can be assembled without interruptions, i.e. all the component types needed by the PCB types of the group are in the feeder of the assembly machine. Finally, the partial setup is a combination of the minimum and unique setup strategy.

1.3 Goals and contribution of this research

This work presents several new problems arising in the field of FMS. These problems are formulated with mathematical precision thus enabling further research of their theoretical properties. The publications of this work show the NP-hardness of the studied problems as well as their relationships to other known problems. This is the first claim of this work –problems we model can not be solved in polynomial time unless NP=P (See [48] for an introduction to computational complexity. The question of whether NP=P or not remains to be unsolved, but it is a generally accepted conjecture that it is not. If so, then these problems can not solved within polynomial time with a Turing equivalent machine.)

We present exact solutions for all deterministic optimization problems. The sixth publication "Scheduling and Management of wearing tools with stochastic lifetimes" studies a complex stochastic optimization problem ([49]) and an analytical solution is reachable only for certain types of probability distributions. The publication presents theoretically valid lower and upper bounds for this problem. In all publications these exact solutions and bounds are used as benchmarks for the heuristic methods. Thus, the main result of this work is that these hard combinatorial optimization problems can be solved in practice nearly optimally using heuristic methods.

We present firm empirical results about the efficiency of the solutions methods in addition to theoretical and procedural discussions. A general approach in the empirical testing is the use of artificially generated test data. This type of data is more challenging for the solution algorithms because they do not have any systematic properties, like certain component combinations in printed circuit boards. Thus, it is not possible to optimize the algorithms for certain type of test data because of the stochasticity involved in the problem instances. Another important property of our empirical testing is the statistical analysis of the results. Although most of the test data is generated through randomization, some runs have also been performed using real production data. In other runs the distribution parameters are chosen so that the generated problem instances resemble real test data.

The structure of all our publications [24,57,58,59,28,29] is summarized to four parts: mathematical formulation, exact solutions and bounds, heuristic solutions methods and empirical testing. The mathematical formulation enables theoretical inspection of the problems. Exact solutions and bounds offer a way to evaluate the efficiency of the heuristic solutions. Heuristic algorithms give a means to solve practical sized problems efficiently. Finally, the empirical testing proves the computational efficiency and quality of results of our heuristic methods.

Chapter 2

Solution methods

The problems we study in this work are combinatorial optimization problems (COP) along with one stochastic optimization problem (SOP). A general description of a combinatorial optimization problem is shown Figure 8. The definition of a COP consists of an objective function *f*, a vector of decision variables ***x*** and a set of constraints on ***x***. A solution of a COP is a *feasible* setting for ***x*** which either minimizes or maximizes the value of the objective function *f*. A setting is said to be feasible if it fulfils the constraints of the definition.

Minimize

$f(\boldsymbol{x})$

Subject to

$g(\boldsymbol{x}) \leq \boldsymbol{b}$

$\boldsymbol{x} \geq 0$

$\boldsymbol{x}$ is discrete

Figure 8. Combinatorial optimization problem.

Although the formulation of Figure 8 does not bind the constraints of the variables or define the type of functions one is allowed to use, the term combinatorial optimization problem is used when the constraint set defines a discrete domain for the decisions variables. In addition, this work studies problems for which the functions are linear, only. A simple example of a COP is the knapsack problem formulated in Figure 9. It is important to notice that these formulations describe an infinite set of real-life problems because the constraints depend on the particular problem *instance* we are solving.

Maximize

$\Sigma_{i=1,\ldots,N}\, x_i v_i$

Subject to

$\Sigma_{i=1,\ldots,N}\, x_i w_i \leq C$

$x_{i=1,\ldots,N} \in \{0,1\}$

Figure 9. An example of a simple combinatorial optimization: knapsack problem. The objective is to maximize the value of the knapsack while respecting the maximum weight *C* the knapsack can hold. There are *N* items, the value of item *i*(=1,…,*N*) is v_i and weight w_i. The variable x_i is 1 if the item *i* is taken in the knapsack, 0 otherwise.

Combinatorial optimization (CO) problems in the field of flexible manufacturing systems are commonly NP-hard, including the knapsack problem. Exact solution methods for practical sized problems require therefore exhaustive computational time. Some of these problems can be solved in polynomial time using approximation algorithms which guarantee that the solution is within certain bounds from the optimal solution. However, the problems in PCB assembly considered this work are hard to approximate [50]. For this kind of problems one must rely on bounding methods and heuristics. Bounding methods are usually relaxations of the original problem. Solving these relaxations tells us that the value of the optimal solution must lie between certain bounds. In general, heuristic methods cannot guarantee anything about the quality of the solution; their efficiency can only be verified through empirical testing.

2.1 Heuristics

The difficulty of finding exact solutions in combinatorial optimization has lead to the wide spread use of heuristics. Heuristic algorithms do not necessarily find an optimal solution and in some cases not even a feasible solution. Numerous empirical studies have however shown that they find near optimal results to many hard COPs.

Heuristic algorithms can be categorized according to the metaheuristic they are based on. The following three subsections briefly describe local search, its' improved version tabu search [51] and genetic algorithms (GA) [52]. Local search and tabu search rely heavily on the idea of 'neighbourhood' while GAs exploit contingency and clever coding of the problem space.

2.1.1 Local search

The first step in local search (also called neighbourhood search) is to find a solution which may be either feasible or not. Once this has been found it can be improved by looking at its' neighbouring solutions. The idea is to iteratively improve the solution by moving from the current solution to a better one in the neighbourhood of the current solution.

A neighbourhood of a solution *S* is defined as a set *N*(*S*) of those solutions which can reached from *S* by doing a simple operation on *S*. Let us consider the discrete knapsack problem defined in Figure 9. In this problem, a candidate solution is a set of items fulfilling the capacity constraint. The neighbourhood of this solution can, for example, consist of those solutions in which one item has been moved out from the knapsack and some other taken in. Figure 10 presents a simple (greedy) heuristic for the knapsack problem. The heuristic first creates some initial solution which can, for example, be a random selection of items as long as the capacity is not exceeded. The algorithm then continues by considering swaps of items and chooses the swap giving the best value for the knapsack while respecting the capacity constraint. This is iterated as long as some improving swap is found.

```
SolveKnapSack($C$, $I$) : set of items                -- C is the knapsack capacity and
    Solution := CreateInitialSolution($C$,$I$);      -- I is the set of items
    do
        $I_o$ := $I \setminus$ Solution;                   -- $I_o$ is the set of items currently not
        BestFound := Solution;                       -- in the solution
        for all $i_o \in I_o$ do
            for all $i \in$ Solution do
                Test := (Solution $\setminus \{i\}) \cap \{i_o\}$;
                If Value(Test) > Value(BestFound) and
                  Weight(Test) $\leq C$ then
                    BestFound := Test;
                end if
            end for
        end for
        if Solution <> BestFound then
            Solution := BestFound;
        else
            BestFound := null;
        end if
    while BestFound  <> null;                       -- Continue as long as the solution
    return Solution;                                -- improves
```

Figure 10. Simple swap heuristic for the knapsack problem.

The heuristic method for the knapsack problem is simple but it shows the most serious flaw of the local search, that is, the convergence to local optimas. The concepts of local and global optimum are illustrated with a simple example in Figure 11. In local search one always looks at the neighbourhood of the current solution and proceeds to the best neighbour. The problem here is that many objective functions have several local optimas, i.e. points where all the neighbours are worse than the current solution. The objective in combinatorial optimization is to find the global optimum which is minimal (maximal) value of the objective function in the whole feasible domain of the decision variables.

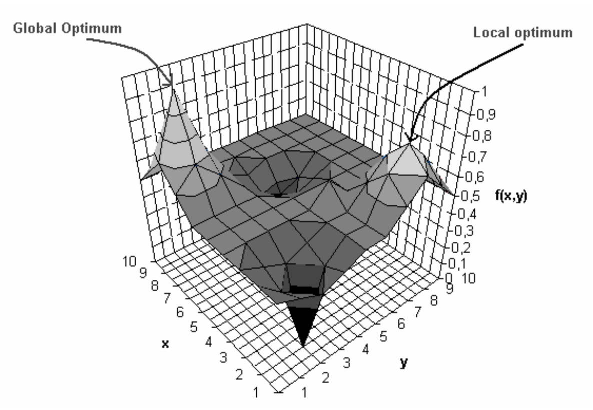


Figure 11. The difference of local and global optimum for two variable function *f*.

2.1.2 Tabu search

The convergence to local optimas in local search can be avoided partially by allowing the search algorithm to move to worse solutions, too. The tabu search (TS) [51] accepts moves to neighbours which are worse than the current solution while at same time maintaining a list of tabu moves which are not allowed in the current state. This list usually contains some recently visited solutions. Roughly speaking, the local search method can be seen as a special case of the tabu search, since its' tabu moves are those which lead to a worse solution than the current one. The best solution is kept in memory because the stopping criterion of the algorithm does not guarantee that the last solution visited by the algorithm is the best solution met during the whole search process.

The key components of a tabu search algorithm are:

- neighbourhood function;
- aspiration criteria;
- tabu list;
- tabu tenure;
- ending criterion; and
- coding of tabu moves.

We explain these components through an example. The local search method for the knapsack problem, which was introduced in section 2.1.1, is shown again in Figure 12, but now it is enhanced with tabu search. The neighbourhood function in the example of Figure 12 is stated explicitly with symbol *N*. The function *N*(*K*,*I*) returns the set of solutions in which some pair of items is swapped between the sets *K* and *I*. The aspiration criteria for this example is that if tabu move leads to a better solution than the best solution found so far, the move is allowed. The tabu list consists of recently tested solutions and the tabu tenure (maximum length of tabu list) is given as parameter *T*. The stopping criterion of our example is the total number of moves, which is given by parameter *M*. Finally, the tabu moves are coded as sets of items. Another possibility to code the tabu moves would be to remember the items involved in the recent swaps. See [51] for detailed discussion of the subject.

```
SolveKnapsackTS(C, I, M) : set of items          -- C is the knapsack capacity and
    Solution := CreateInitialSolution(C,I);      -- I is the set of items
    NumberOfMoves := 0;
    do
        $I_o$ := I \ Solution;                    -- $I_o$ is the set of items currently not
        BestFound := null;                       -- in the solution
        for all S $\in$ N(Solution, $I_o$) do
            if Weight(S) $\leq$ C and (not Has(Tabulist, BestFound) or
              Value(S) < Value(Solution)) then
                if BestFound = null then
                    BestFound := S;
                else if Value(S) > Value(BestFound) then
                    BestFound := S;
            end if
        end for
        if Solution <> BestFound then
            Append(Tabulist, Solution);
            if Length(Tabulist) $\geq$ T then
                RemoveFirst(Tabulist);
            end if
            Solution := BestFound;
        else
            BestFound := null;
        end if
        NumberOfMoves := NumberOfMoves + 1;
    while NumberOfMoves $\geq$ M;
    return Solution;
```

Figure 12. Tabu search method for the knapsack problem.

2.1.3 Genetic algorithms

The evolution of living organisms has been an inspiration to the development of genetic algorithms (GA). The idea of GAs is to create an initial population of individuals, each individual describes a solution to the problem to be solved. Every individual is evaluated during each iteration using a suitable fitness function. The value of this function tells how good the individual is (with respect to the goal of the problem to be solved). The key components of a GA are coding an individual, fitness evaluation, crossover, mutation and selection. These components are discussed next and after that a brief description of the main phases of a GA is given.

The coding of an individual is done similarly as a gene codes the function of living things. A single gene then describes some part of the solution. The gene itself can be a number, letter or something more complex. The only limitation to the coding is that a GA must be able to merge the genes of two individuals so that the result is still a valid solution to the problem. Coding in our publications is based on the idea of the problem space search [62]. In this technique an individual is simply a vector of real numbers, which is interpreted by some algorithm translating the vector to the corresponding solution of the original problem.

The fitness of an individual is evaluated by using a function which translates the solution described by the individual to a number. For example, in a process control application, the result of this function can be the average processing time of a printed circuit board.

In the crossover, GA selects two individuals (parents) and merges their genes. The selection of the parents can be done in several ways, but the GAs of the present study use the tournament selection method. The number of tournament rounds is selected randomly in our GAs from an interval given as parameter. During each tournament round candidate parents are selected randomly from the population and compared against each others. For example, if the number of tournament rounds is 1 then the selection of the parents is purely random. After the selection of the parents, GA performs the actual crossover. In its simplest form crossover takes half of the genes from other parent and the rest from the other. The combination of these halves forms an offspring which is included in the next generation.

It is common that each individual of the population is mutated with a small probability during each iteration. The simplest form of the mutation is a replacement operation in which one gene of an individual is replaced with a random value gene.

A genetic algorithm begins by creating an initial population of a certain size (given as a parameter). This population is commonly created purely random. In the evolution phase GA selects the individuals for the next generation through direct transfer and crossover as described above. Some percentage of the population is transferred directly to the next population and the rest are discarded. In order to keep the population size constant, a number of offsprings are created through crossover. After the creation of the next generation each individual is allowed to go through a mutation with a certain probability. The evolution phase is repeated a given number of iterations and the result of the whole process is the solution described by the individual with the best fitness value after the last iteration. See [52] for more details on the general structure of GAs.

2.2 Bounding methods

Algorithms used to solve mixed integer linear programs rely mainly on the branch-and-bound-method which uses heavily lower and upper bounds to make the search converge more quickly. Lower bounds can also be used to estimate how close to the optimal solution some heuristics can get.

The idea in lower bounding is to relax some of the problem constraints to make it easier to solve and in this way solve problems of practical size. Good lower bounds are therefore very important in solving combinatorial optimization problems.

2.2.1 Linearization

The combinatorial optimization problems studied in this work contain both discrete and continuous variables. Many efficient solution algorithms, like the simplex algorithm, are known for problems which have only continuous variables. Solving the mixed and discrete optimization problems is known to be an NP-hard problem and therefore usually only toy-sized problems can be solved exactly.

In linearization of an optimization problem one simply replaces all discrete variables with continuous variables, this is clearly a relaxation since the variables have now a greater domain. The problem with this approach is that the lower bounds found in this way are usually not tight and therefore have only a little practical use.

2.2.2 Lagrangean relaxation

This method is based on the duality of a mixed integer linear programs (see [53]). The basic idea is simple, some of the constraints are added to the objective function as a penalty if they are broken. The amount of penalty depends on the Lagrangean multiplier which is associated to each constraint which is moved to the objective function. It can be proved that Lagrangean relaxation produces always a lower bound to the original problem.

There are two problems in this type of relaxation. First, we must find good Lagrange multipliers in order to get a tight lower bound. These multipliers are usually found in an iterative manner, for example by using the subgradient optimization technique (see [53]). The second problem is that some optimization problems are inherently difficult, meaning that one must relax so many constraints of the problem that the lower bounds found are not tight enough. In the worst case scenario the lower bound can be worse than the one found through linear relaxation which is much simpler to implement and very fast to calculate.

2.3 Exact methods

Finding exact solutions to many different combinatorial optimization problems has turned out to be very difficult in practice. The problems we study in this work are all NP-hard, which means that there are no polynomial time solution algorithms known for them. A generally accepted conjecture is that there will never be polynomial time algorithms to solve these problems exactly (for all possible problem instances), although it is an open question in mathematics.

2.3.1 Enumeration

The combinatorial optimization problems we study have a finite number of solutions. Thus, the simplest way to solve such a problem is to enumerate all the possible solutions and choose the one giving an optimal solution. It should be noted, however, that the number of possible solutions is usually impractically large. For example, one is given a function *f* and some set of items and the task is to find the order of items giving the lowest value for *f*. Now, if the number of jobs is 20 then there are 20! different possible solutions. Let us further assume that the computer we are using to solve this problem can evaluate 1 000 000 solutions per second. Then it would take over 70 000 years to solve this problem. The problems we study in this work include orderings with more than 100 items.

2.3.2 Branch-and-bound

Production planning problems studied here can be formulated as mixed integer linear programs (MILPs). A general form of a MILP is given in Figure 13. MILPs form a special class of the general combinatorial optimizations problems formulated in Section 2. A mixed integer linear program (MILP) consists of discrete and continuous variables, parameter values presented by real numbers, a linear objective function to be minimized and some linear constraints for the variables. The branch-and-bound method is a general solution method for optimization problems, but we present it in the context of solving MILPs. This is justified because all problems or their bounding methods discussed here will be formulated as MILPs.

Minimize

$$\Sigma_{i=1,\ldots,N}\, a_i x_i + \Sigma_{j=1,\ldots,M} b_i y_j$$

Subject to

$$\begin{array}{ll} x_i \in \{0,1\} & \text{for all } i=1,\ldots,N \\ y_j \geq 0 & \text{for all } j=1,\ldots,M \\ \Sigma_{i=1,\ldots,N}\, c_{ir} x_i \leq d_r & \text{for all } r=1,\ldots,L \\ \Sigma_{j=1,\ldots,M}\, e_{js} y_j \leq f_s & \text{for all } s=1,\ldots,K \end{array}$$

Figure 13. The general form of a mixed integer linear program (MILP). In mixed models we have two sets of variables; one set of variables (x) is constrained to have only values in {0,1} and the other set (y) is constrained to positive real numbers. The relation of matrices C and E multiplied by the variable values x and y force additional linear constraints for these variables.

The problem of solving a general MILP is an NP-hard problem and it is possible that the search for the optimal solution with the branch-and-bound method takes too long time. In Figure 14 we have a general the branch-and-bound [54] method for MILPs. It forms a search tree in which each node fixes later some discrete variable and forms one branch for each possible value of the discrete variable. Once we have reached a leaf and fixed all the possible discrete variables we get an upper bound for the solution. This upper bound is then used as a pruning method to prevent search from branching to directions for which the lower bound is larger than the best current solution. The lower bound for a certain branch is calculated from the linear relaxation of the non-fixed variables. The linear relaxation of a MILP is presented in Section 2.3.1. At some point of the search there is only one solution left, because the method has searched through the whole tree. The time required to do this depends mostly on how well the pruning works, which again depends on the problem in question and the choice of the order for fixing the values of the variables. In many cases creating a problem specific branch-and-bound method yields better results than using this general method because the variables can then be fixed in such an order that the search converges more quickly than in this general approach.

```
BranchAndBound($C, f, X, Y, UB$) : solution value -- $C$ set of constraints, $f$ function to be
                                                  -- minimized $X$ set of discrete variables,
    $OptimalLP$ := SolveLP($C, f, X \cap Y$);     -- $Y$ set of continuous variables and
                                                  -- upper bound for the solution
    if $OptimalLP > UB$ then Return $UB$;
    else if for all $x \in X \mid x \in \{0,1\}$ then
        if $UB < OptimalLP$ then                  -- Integer solution was found, checking if
            $UB := OptimalLP$;                    -- it is better than the current best
    else                                          -- solution
        $x := Random(x \in X \mid x$ not integer);    -- All 0/1-variables do not
                                                  -- have an integer value,
        $UB_1$ := BranchAndBound($C \cap \{(x = 0)\}, X, Y, UB$);    -- therefore one of them is
        $UB$ := BranchAndBound($C \cap \{(x = 1)\}, X, Y, UB_1$);    -- fixed to either 0 or 1
    end if

    return $UB$;
```

Figure 14. Branch-and-bound method for MILPs.

2.4 Optimization software

There are numerous optimization software packages available which can be categorized roughly to the general and application specific software. Software packages used for solving the MILP models are general since they do not exploit any problem specific knowledge in their implementation. We used ILOG Solver [55] to implement the MILP models. It is one of the most efficient software packages available to linear programming and mixed integer linear programming. The solution algorithm in it uses the branch-and-bound-method enhanced with some "rule of thumb" knowledge. The models are implemented with OPL-language which resembles somewhat traditional procedural languages, like the C-language. The definition of the OPL-language contains also a scripting language. Scripting makes it possible to create higher level system, like sub-gradient optimization, with the optimizer which we used in one of our relaxations. The ILOG solver has a graphical interface and many techniques to visualize the solution and the search for the solution. For example the search tree used by the branch-and-bound method can easily be visualized.

Application-specific optimization software uses problem-specific knowledge in searching for good solutions. The way they use the problem specific knowledge depends on the type of the software – some software systems only seek to improve the existing feasible solutions. In this case heuristic algorithms which are mainly based on "rules of thumbs" are used. In this type of software there is no guarantee of how good the final solution will be with respect to the optimal solution. Because heuristic algorithms are usually fast and can find good solutions, this type of exploitation of problem specific knowledge is quite common. The other way of using the knowledge is to enhance some algorithm which guarantees an optimal solution to, for example, limit the search space. The branch-and-bound method, for example, can be enhanced by choosing the order of fixing the variables in a way which leads to a smaller search tree for some particular problem.

The flexible manufacturing systems, which we study in this work, usually have some vendor-supplied software with them which is based on heuristic methods. There are, however, so called third party software packages which support several vendor machines in single software. In the field of PCB assembly Trilogy 5000 [56] is an example of such a software package.

Chapter 3

Summary of publications

This section briefly summarizes the six publications of this work. The first three publications deal with the PCB assembly and especially the job grouping problem arising in the single machine optimization. The fourth publication is also about PCB assembly although it can be applied to other flexible manufacturing systems as well. It models a problem which considers the optimization of PCB assembly from another perspective, i.e. instead of grouping the PCBs it considers a subproblem of ordering the PCBs.

The results of the latter two publications can be applied widely in the field of flexible manufacturing systems. They consider the issue of tool wearing which has been largely omitted in the field. Only a couple publications exist which model the tool wearing and scheduling problem integrated as we do. The fifth publication considers a complicated scheduling problem in CNC production in which the tool lifetimes are deterministic. The last publication still relaxes the assumption of deterministic lifetimes for the tools and assumes that they follow some probability distribution. Although these two papers consider different problems, these problems have a common objective which is to minimize the average completion time of the given job set.

3.1 Grouping PCB assembly jobs with typed component feeder units

Most studies concerning the job grouping problem assume that the assembly machine is equipped with one linear feeder. The present study [57] generalizes the job grouping problem (JGP) by relaxing the assumption of a single feeder. Motivation for studying this problem comes from the recent advances in PCB assembly machine technology. Some new machines are able to use several feeders simultaneously which allows the use several different feeder types, for example tape, tray and track feeders. This makes PCB printing more flexible since it is then possible to use more different types of components during a single printing.

We formulate the new problem, called job grouping with typed feeder units (JGP-T) and show that the basic JGP is a special case of the JGP-T. This proves that the JGP-T is also NP-hard. Next we discuss the relation of the JGP-T to other problems and especially point out that there are similarities between clustering and JGP-T.

The problem formulation is also translated to an integer programming formulation which is later implemented with ILOG solver. Four new fast and efficient heuristic algorithms are given to solve the JGP-T. They are based on the algorithms developed earlier for the basic JGP. The new algorithms differ with respect to the similarity measures used. Finally, the efficiency of the heuristics is compared against optimal solutions for small problem instances. For large problem instances we use a naïve approach to quantify the benefit gained from using complex heuristics.

We conclude that the JGP-T is hard to solve exactly and therefore there is a true need for the heuristics developed in this paper. The empirical testing also shows the superiority of the tabu search over the other local search methods. The testing against the optimal solutions also shows that tabu search method gives near optimal results for small problem instances.

3.2 Grouping of PCB assembly jobs in the case of flexible feeder units

The most advanced PCB assembly machines are equipped with a feeder which is simply a holder for subfeeders (called bins). These bins have different restrictions for the component types they can accommodate. Further, the feeder sets some restrictions to the use of bins, like the maximum number of bins and maximum width for the bins. The basic JGP formulation can not model problem as complex as this with reasonable accuracy and therefore this paper [58] defines the job grouping problem with flexible feeder units (JGP-B) and introduce several methods for solving it.

The NP-hardness of the JGP-B follows from the NP-hardness of the basic JGP. The JGP-B formulation and its' integer programming translation includes a very large number of discrete variables. It is therefore not a surprise that exact solutions are found for small problem instances, only. We therefore develop heuristic methods for the problem. The heuristics used for the JGP-T and the basic JGP offer a good starting point for the development of these methods. A new aspect in the JGP-B is the modifiable feeder unit which complicates the solution methods tremendously. This is due to the fact that checking the feasibility of a group is not a simple test as in the previous variants of the

JGP. The feasibility check is now an NP-complete problem in itself, which follows from the NP-completeness of the bin packing –problem. Because of the complexity of the feasibility checking one must resort on heuristics on this part of the problem as well. We introduce two heuristic methods to the feasibility checks.

The empirical results show that the integer programming formulation which is implemented on the ILOG solver is somewhat useless in practice because it is able to solve toy-sized problems only. We therefore test the efficiency of our heuristics against a naïve approach and against each other. The naïve approach is the kind of solution a human operator might be able to use.

Our conclusion is that the heuristic methods we develop are useful in practice because they give superior results over the naïve method. The testing also shows that the exact solution of JGP-B is hard for problem instances of practical size.

3.3 A general approach to grouping of PCB assembly jobs

The previous studies concerning the JGP produced a number of variant specific algorithms and definitions. This paper [59] gathers the different variants of the JGP under a single formulation. First we formulate three different variants of the JGP. After that, we introduce a general MILP programming formulation which solves the three variants of the JGP when they are transformed to general form according to the algorithm given with the MILP formulation.

The MILP model is followed by a short description of heuristic methods used earlier to the different variants. After this, we developed two new variant independent heuristic algorithms to solve the general JGP. These methods are tested against the best problem specific methods in the empirical testing section. We also test the limits of exact solving by implementing the general MILP formulation on ILOG solver.

Our conclusion is that the general solution algorithms perform equally well as the problem specific algorithms. The general MILP formulation is able to solve problems of the same size as the problem specific models when certain additional constraints are set. The better one of our two new heuristic algorithms solves the different JGP variants in some cases even better than the previous problem specific methods.

3.4 The general two level storage management problem

Another approach to the optimization of PCB assembly is to order the PCB assembly tasks so that the total number of component switches between PCB types is minimal. An important subproblem for this so called tool switching problem is the tool loading -problem. In the tool loading –problem the order of PCB batches is fixed and the objective is to minimize the number of tool switches by making the tool management decisions intelligently.

The keep tool needed soonest (KTNS) –rule gives an optimal solution to the tool loading -problem if all the tools (or in the case of PCB assembly components) use the same amount of capacity from the tool magazine. The previous studies concerning this problem have either omitted the different sized tools or the physical placement of the tools. We extend [24] these models by considering tools (components) of different widths and by modelling their physical placement to the tool magazine. The formulation of the problem is followed by a NP-hardness proof.

We actually study two different versions of the general two level storage management problem. In the first one we assume that each job requires only one tool during its processing. This is applicable to some FMSes but not to the PCB assembly. This is why we also present solution methods for the case where jobs can be processed with several tools each. For the single-tool case we propose an algorithm which is based on the ideas of Matzliach and Tzur [60]. For the multi-tool case we introduce two new algorithms.

The integer programming formulation for our problem is somewhat complex and the implementation of that formulation is not able to solve problems of practical size, even small problems turn out be difficult. To test the efficiency of the heuristics we point out that if we omit the physical placement of the tools we get a lower bounding method for the problem. We therefore use the integer programming formulation of Matzliach and Tzur to calculate lower bounds for our problem instances.

The heuristics are compared empirically against random, naïve and lower bound methods. The results show that the new heuristics are superior over the naïve and random methods. They also perform very well with respect to the lower bounds.

3.5 Job ordering and management of wearing tools in flexible manufacturing

This paper [28] formulates a scheduling problem arising in the FMSes. In the *scheduling with wearing tools* (SWT) –problem one must create a production plan which minimizes the average completion time of the given jobs. It is supposed that the CNC machine uses limited life-time tools to process the jobs. The contribution of this research is a new way of modelling the tool magazine. Previous studies concerning this problem have assumed that the machine uses only tools of one tool type to process parts. We extend this model by allowing the use of several tool types and a limited capacity of the magazine.

A traditional approach to tool management and scheduling has been to omit the completion times of the jobs from the minimization objective and to concentrate on the minimization of the number of tool switches. This traditional approach assumes that the job processing times are dominated by the tool switching times. Technological advances of the CNC machines have, however, reduced the switching times considerably thus giving additional motivation for studying the SWT-problem. Another assumption has been that tool switches occur due to part mix. However, studies have shown that tool switches occurring because of tool wear may be much more likely than the ones occurring due to part mix [61].

This paper begins with a mathematical formulation of the SWT-problem which is followed by NP-hardness proof and an integer programming solution. The IP-solution needs a large number of discrete variables and is thus unusable in practice expect for small problem instances. We therefore present three heuristics to solve the problem: a simple local search, a genetic algorithm and a combination of these. These three methods all use tool switch policy (TSP) which we develop first. The SWT-problem is solved in two steps. In the first step we fix some order and in the second step we calculate the average cost for this order. For example, the local search method makes some changes to the job processing the order of the jobs and re-calculates the average cost. The same is repeated as long as the average cost decreases.

Because this problem is new, we consider two different relaxations to it in order to determine lower bounds for the solutions. The linear relaxation is used as a comparison method to the Lagrangean relaxation. The results are somewhat disappointing, since the Lagrangean relaxation with subgradient optimization is not able to find significantly better lower bounds than the simple linear relaxation.

The empirical results include four test runs. The purpose of the first run is to find out how large problems can be solved exactly by implementing the IP-model on the ILOG solver. The second test is used to fine-tune our heuristics. The third test compares the heuristic solutions against optimal solutions. The final test compares the efficiency of the heuristics against each other and a naïve solution for large problem instances.

We conclude that exact solving of the SWT-problem is limited to small problems only. Lower bounding through Lagrangean relaxation is not able to produce high quality bounds in reasonable time: the results are not significantly better than those by linear relaxation. We therefore find the SWT-problem hard to relax.

3.6 Job scheduling and Management of wearing tools with stochastic lifetimes

This paper [29] considers a stochastic optimization problem which is an extension to the scheduling with wearing tools (SWT) –problem considered in the previous paper. In this problem, called Job scheduling with stochastic tool lifetimes (JSSTL), we are given a set of jobs to be processed on a single CNC machine. The jobs are processed with tools which have limited life-times. In the previous paper we assumed that these life-times depend only on the type of the tool. It is assumed in the present study that the life-times vary between different tools of same type following some probability distribution. This brings additional difficulty to the formulation because a tool may break down during the processing of a job. In this case the unfinished job must be discarded and started from the beginning. The problem now calls for a schedule which minimizes the *expected* average processing time of the jobs. The actual realized average processing time depends on the physical tools used to process the jobs.

The problem definition is followed by an example and a discussion about the lower bounding. The lower bounding method works only if certain assumptions about the probability distributions of the lifetimes are made. The first assumption is that the tools must last at least certain time which is longer than any of the jobs require in processing. One must also assume that the tools have some upper limit to their life-time after which the tool must be changed. With these assumptions the IP-model of the SWT-problem can be used to calculate the lower bounds of the JSSTL-problem.

Exact solution of this problem seems to be extremely difficult or impossible. This is due to the fact that there are an infinite number of tool realization combinations which can occur. The only possibility would be to devise an integral over the possible life-time realizations. Even for much simpler stochastic optimization problems these analytical solutions have been tedious and in some cases not even possible. It is hence highly unlikely that such a solution would be possible for the JSSTL-problem.

Our solution to this problem is the use of a genetic algorithm which solves the problem in two parts. The orderings of the jobs are coded as individuals and the fitness function calculates an approximation of the expected average processing cost (EAPC) for the individuals. The fitness function approximates the EAPC by simulating the job processing runs. The number of samples varies between 100 to 2000 depending on the size of problem. As for the deterministic version of this problem, the fitness function employs a tool switch policy to make the tool management decisions while evaluating the cost for a certain order and realizations of the tools. Our tool switch policy (TSP) for this problem uses a greedy heuristic which attempts to minimize the expected processing time of the next job.

One of the major goals of our empirical testing was to find out how many sampling runs are required in order to get the variance of the EAPC low enough for later evaluation of the efficiency of our GA. Another major goal was to compare the GA solutions to lower bound and upper bounds. Finally, we evaluated the value of stochastic information by comparing a TSP which uses expected life-times and the TSP designed specifically to exploit the information about the probability distributions of the tools.

We concluded that an analytical solution to the problem we study seems impossible due to the complexity of the problem. The bounding methods we developed give tight bounds for small problem instances. The GA along with the TSP gives near optimal results to small problem instances. Also, the value of stochastic information which we approximated in the empirical testing is substantial.

Chapter 4

Conclusions

The studies concerning tool switching in FMSes assume usually fixed sized tools. It is, however, common that in many real world applications, like PCB assembly, tools (in this case component feeders) have variable widths. This feature of the problem is modelled in three publications of this work concerning PCB assembly. In addition, they model the assembly machines more accurately than previous studies concerning the job grouping problem (JGP). The first publication [57] which studies typed feeder units, models a problem in which components can have variable widths and variable feeder types (JGP-T). This makes the modelling more accurate because it is common nowadays that the machines can use several different packaging techniques (feeder types) for the components. We showed that this problem is NP-hard and give a mathematical formulation. The algorithms we devised were based on the previous studies considering the simpler variants of this problem. Comparison to exact solutions clearly indicated the efficiency of our algorithms.

In the second publication [58] we modelled an even more complex assembly machine in which the tool magazine (feeder) has a structure that can be changed quickly (JGP-B). This extension to the JGP makes the problem a lot more complicated and exact solutions can be calculated only for very small problem instances. The algorithms we developed were compared against naïve methods.

In the third publication [24] we studied the two level storage management problem with variable width tools. Previous studies concerning this problem have not modelled the physical placement of the tools to the magazine. Our problem formulation models the tool magazine physically accurately and therefore the solutions of our algorithms can be used directly in real world applications. We also showed that this problem is NP-hard and formulated it as a MILP. We devised several heuristic algorithms for the two variations of the problem and compared them against lower bounds. The comparison as well as running time analysis showed that these algorithms find near optimal results in an acceptable time.

The wearing tools problem in flexible manufacturing systems has been largely omitted in the studies concerning tool switching. Only a few studies addressing this problem exists although tool wearing has been shown to be an important issue in FMSes. Most studies consider tool switching occurring due to the part mix, but it has been reported that tool switching occurring due to tool wearing can be much more frequent [61].

We modelled two new problems concerning tool wearing in FMSes. The first one [28] extended an existing study [27] which assumed one tool type. Our formulation allows several tool types and a limited capacity magazine. This problem is applicable to many of the existing CNC machines which are a central part of flexible manufacturing systems. The latter publication [29] generalized the problem further and instead of deterministic tool life-times it assumed that the life-times follow some probability distributions.

Both of the wearing tools problems were solved using genetic algorithms. The coding of a solution was based on problem space search [62]. We showed that these algorithms give near optimal results by comparing them exact solutions and lower bounds. The exact solution was possible to the deterministic version of the problem, only. For the stochastic problem we were only able to solve lower bounds. It turned out that using stochastic information in GA gives a significant advantage over the expected life-time solution.

Despite of the increase in computational power and the development of new algorithms for solving MILP models there remains still a true need for problem specific heuristics as shown by our research. In many complex combinatorial optimization problems the known metaheuristics find near optimal results. Especially, the genetic algorithms can be applied to a wide variety of combinatorial optimization problems when the input is coded in a suitable way.

Chapter 5

Bibliography

[1] Groover M.P., *Automation, production systems and computer-integrated manufacturing*. Prentice-Hall, 2000.

[2] Rehg J.A., *Computer integrated manufacturing*. Prentice Hall College Division, 1994.

[3] Greenwood F., *Introduction to computer integrated manufacturing*. Harcourt College Publications, 1989.

[4] Wenqi H. and Aihua Y., An improved shifting bottleneck procedure for the job shop scheduling problem. *Computers & operations research*, pp. 2093-2110, 31, 2004.

[5] Potts C.N. and Whitehead J.D., Workload balancing and loop layout in the design of a flexible manufacturing system. *European journal of operational research*, pp. 326-336, 129, 2001.

[6] Persi P., Ukovich W., Pesenti R. and Nicolich M., A hierarchic approach to production planning and scheduling of a flexible manufacturing systems. *Robotics & computer integrated manufacturing*, pp. 373-385, 15, 1999.

[7] Berardi V.L, Zhang G. and Offodile O.F., A mathematical programming approach to evaluating alternative machine clusters in cellular manufacturing. *International journal of production economics*, pp.253-264, 58, 1999.

[8] Noble P. and Moore R., *Assembly of printed circuit boards*. Kluwer academic publishers, 1994.

[9] http://www.samsungtechwin.com/default.asp

[10] Johnsson M., Operational and tactical level optimization in printed circuit board assembly, TUCS Dissertations, No. 16, 1999.

[11] Aneja Y.P. and Punnen A.P., Multiple bottleneck assignment problem. *European journal of operational research*, pp.167-173, 112, 1999.

[12] W. Luggen, *Flexible Manufacturing cells andsSystems*, Prentice-Hall Inc., Upper Saddle River, New Jersey, 1991.

[13] Anderson A.L., *Microstation J: An introduction to computer-aided design*. Schroff Development Corporation, 2001.

[14] http://www.eikoncnc.com/eikonmvl.htm

[15] D'Angelo A., Gastaldi M. and Levialdi N., Multicriteria evaluation model for flexible manufacturing system design. *Computer integrated manufacturing systems*, pp. 171-178, 9(3), 1996.

[16] Devaraj S., Hollingworth D.G. and Schroeder R.G., Generic manufacturing strategies: an empirical test of two configurational typologies. *Journal of operations management*, pp. 427-452, 19, 2001.

[17] Waller S., Criteria for selecting control systems in flexible manufacturing. *Robotics & computer integrated manufacturing*, pp. 81-88, 7(1), 1990.

[18] Reeves C.R., *Modern heuristic techniques for combinatorial optimization*, McGraw-Hill International, 1995.

[19] Crama Y., Kolean A.W.J., Oerlemans A.G. and Spieksma F.C.R., Minimizing the number of tool switches on a flexible machine. *The International journal of flexible manufacturing systems*, pp. 33-54, 6, 1994.

[20] Al-Fawzan M.A. and Al-Sulta K.S., Tabu search based algorithm for minimizing the number of tool switches on a flexible machine. *Computers & industrial engineering*, pp. 35-47, 44(1), 2003.

[21] Djellab H., Djellab K. and Gourgand M., A new heuristic based on a hypergraph representation for the tool switching problem. *International journal of production economics*, pp. 165-176, 64(1-3), 2000.

[22] Bard J.F., A heuristic for minimizing the number of tool switches on a flexible machine. *IIE Transactions*, pp. 382-391, 20(4), 1988.

[23] Tzur M. and Altman A., Minimization of tool switches for a flexible manufacturing machine with slot assignment of different tool sizes. *IIE Transactions*, pp. 95-110, 36, 2004.

[24] Hirvikorpi M., Salonen K., Knuutila T. and Nevalainen O.S., The general two level storage management problem: a reconsideration of the KTNS-rule. To appear in *European journal of operational research*.

[25] Tang C.S. and Denardo E.V., Models arising from a flexible manufacturing machine, part I: minimization of the number of tool switches. *Operations research*, pp.767-777, 36, 1988.

[26] Brassard G. and Bratley P., *Fundamentals of algorithmics*. Prentice-Hall Incorporated, 1996.

[27] Akturk M.S., Ghosh J.B. and Gunes E.D., Scheduling with tool changes to minimize total completion time: a study of heuristics and their performance. *Naval research logistics*, pp. 15-30, 50, 2003.

[28] Hirvikorpi M., Knuutila T. and Nevalainen O., Job ordering and management of wearing tools in flexible manufacturing, to appear in *Engineering optimization*, 2004.

[29] Hirvikorpi M., Knuutila T,, Leipälä T. and Nevalainen O.S., Job Scheduling and Management of Wearing Tools with Stochastic Lifetimes, Submitted for publication.

[30] Blazewicz J., Eiselt H.A., Finke G., Laporte G. and Weglarz J., Scheduling tasks and vehicles in a flexible manufacturing system. *The international journal of flexible manufacturing systems*, pp. 5-16, 4, 1991.

[31] Blazewicz J. and Finke G., Scheduling with resource management in manufacturing systems. *European journal of operational research*, pp. 1-14, 76, 1994.

[32] Sethi S.P., Sriskandarajah C., Sorger G., Blazewick J. and Kubiak W., Sequencing of parts and robot moves in a robotic cell.

The international journal of flexible manufacturing systems, pp. 331-358, 4, 1992.

[33] Crama Y., Combinatorial optimization models for production scheduling in automated manufacturing systems. *European journal of operational research*, pp. 136-153, 99(2), 1997.

[34] Agnetis A., Pacciarelli D. and Rossi F., Lot scheduling in a two-machine cell with swapping devices. *IIE Transactions*, ppp. 911-917, 28, 1996.

[35] Crama Y. and van de Klundert J., Cyclic scheduling of identical parts in a robotic cell. *Operations research*, pp. 65-82, 52(1), 2004.

[36] King J.R., Hodgson T.J. and Chafee F.W., Robot task scheduling in a flexible manufacturing cell. *IIE Transactions*, pp. 80-87, 25, 1993.

[37] Kise H., On an automated two-machine flowshop scheduling problem with infinite buffer. *Journal of the operations research society of Japan*, pp. 354-361, 34, 1991.

[38] Liu S.C. and Lin L., Dynamic sequencing of robot moves in a manufacturing cell. *European journal of operational research*, pp. 482-497, 69, 1993.

[39] Sriskandarajah C., Hall N.G., Kamoun H. and Wan H., Scheduling large robotic cells. Working paper, College of business, The Ohio state university, 1994.

[40] Armstrong R., Lei L. and Gu S., A bounding scheme for deriving the minimal cycle time of a single-transporter N-stage process with time-window constraints. *European journal of operational research*, pp. 130-140, 78, 1994.

[41] Hanen C. and Munier A., Periodic scheduling of several hoists. *Proceedings of the fourth international project management and scheduling conference*, pp. 108-110, 1994.

[42] Lei L. and Wang T.J., The cyclic hoist-scheduling problem is NP-complete. *Working paper 89-16,* Graduate school of management, Rutgers university, 1989.

[43] Phillips L.W. and Unger P.S., Mathematical programming solution of a hoist scheduling program. *AIIE transactions*, pp. 219-225, 8, 1976.

[44] Jaikumar R., van Wassenhove L.N., A production planning framework for flexible manufacturing systems. *Journal of manufacturing and operations management*, pp. 52-79, 2, 1989.

[45] Stecke K.E., Formulation and solution of nonlinear integer production planning problems for flexible manufacturing systems. *Management science*, pp. 273-288, 29, 1983.

[46] Stecke K.E., Algorithms for efficient planning and operations of a particular FMS. *The international journal of flexible manufacturing systems*, pp. 287-324, 1, 1989.

[47] Stecke K.E. and Talbot F.B., Heuristics for loading flexible manufacturing systems. *Flexible manufacturing: recent deveploments in FMS, Robotics, CAD/CAM, CIM*, pp. 73-85, 1985.

[48] Sipser M., *Introduction to the theory of computation*. Course technology, 1996.

[49] Birge J.R. and Louveaux F., *Introduction to stochastic programming*. Springer-Verlag, 1997.

[50] Crama Y. and van de Klundert J., Worst-case performance of approximation algorithms for tool management problems. *Naval research logistics*, pp. 445-462, 46(5), 1999.

[51] Glower F., Glower F.W. and Manuel L., *Tabu search*. Kluwer academic publishers, 1998.

[52] David E. Goldberg, *Genetic algorithms in search, optimization and machine learning*.
Addison-Wesley, 1989.

[53] Pourbabai B., Design of a flexible assembly line to control the bottleneck. *Computer integrated manufacturing systems*, pp. 122-133, 7(2), 1994.

[54] Papadimitriou C.H. and Steiglitz K., *Combinatorial optimization: algorithms and complexity*. Dover publications, 1998.

[55] http://www.ilog.com/products/solver/

[56] http://www.valor.com/

[57] Knuutila T., Hirvikorpi M., Johnsson M. and Nevalainen O.S., Grouping PCB assembly jobs with typed component feeder units. *International journal of flexible manufacturing*, 16(2), 2004.

[58] Knuutila T., Hirvikorpi M., Johnsson M. and Nevalainen O.S., Grouping of PCB assembly jobs in the case of flexible feeder Units. *Engineering optimization*, 37(1), 2005.

[59] Hirvikorpi M., Johnsson M., Knuutila T. and Nevalainen O.S., A General approach to grouping of PCB assembly jobs. To appear in *International journal of computer integrated manufacturing*, 2003.

[60] Matzliach B. and Tzur M., Storage management of items in two levels of availability, *European journal of operational research*, pp. 363-379, 121, 2000.

[61] Gray E., Seidmann A. and Stecke K.E., A synthesis of decision models for tool management in automated manufacturing. *Manage Sci*, pp. 549-567, 39, 1993.

[62] Storer R.H., Wu D.S and Vaccari R., New search spaces for sequencing problems with application to job shop scheduling. *Manage Sci*, pp. 273-288, 29, 1983.

Chapter 6

Original publications

Publication I

Knuutila T., Hirvikorpi M., Johnsson M. and Nevalainen O.S., Grouping PCB assembly jobs with typed component feeder units. *International Journal of Flexible Manufacturing*, 16(2), 2004.

The International Journal of Flexible Manufacturing Systems, 16, 151–167, 2004

Grouping PCB Assembly Jobs with Feeders of Several Types

TIMO KNUUTILA* knuutila@cs.utu.fi
MIKA HIRVIKORPI
Turku Centre for Computer Science, University of Turku, Lemminkäisenkatu 14 A, SF-20520 Turku, Finland

MIKA JOHNSSON
Valor Computerized Systems, Ruukinkatu 2, 20540 Turku, Finland

OLLI NEVALAINEN
Turku Centre for Computer Science, University of Turku, Lemminkäisenkatu 14 A, SF-20520 Turku, Finland

Abstract. A variant of the classical job grouping problem (JGP) in printed circuit board (PCB) assembly is considered. Studies on JGPs have assumed a single feeder from which the components are retrieved and then placed on the PCB. Recent advances in technology have made it possible to use several different kinds (types) of feeders at the same time. In a JGP, the aim is to group the PCBs so that the cardinality of the grouping is minimal and each group can be processed without rearranging the contents of the feeder. In the *job grouping problem with several feeder types* (JGP-T) the goal is the same but instead of one linear feeder we have several feeders and each component is associated with a given feeder type which restricts its placement. We give a mathematical formulation for the JGP-T and show that it is hard to solve to optimality for problems of practical size. The connections of JGP-T to known problems are discussed. We also propose several efficient heuristics and compare their results against optimal solutions.

Key Words: combinatorial optimization, electronics assembly, heuristics, job grouping

1. Introduction

The traditional setting of the *job grouping problem* (JGP) assumes that we have a set of *part types* (or jobs) to be processed on a flexible manufacturing machine, and the processing of each such part type requires that certain *tools* have been installed on the machine. In the context of printed circuit board (PCB) assembly, a 'part type' corresponds to a certain PCB type, and the 'tools' are the electronic components to be installed on the board. For efficiency reasons, PCBs are typically processed as batches of the same PCB type, and each such batch is considered as a component assembly job. Since different board types usually require different components, and the assembly machinery has certain capacity for them, the setting of the machine must be changed from time to time in order to manufacture all PCBs in a production plan. This change of tools or components, is called a *setup occasion*.

*To whom correspondence should be addressed.

The number of setup occasions should be small, since they are time-consuming in practice (Tang and Denardo, 1988). This motivates the job grouping problem, where we want to partition the set of part types into a minimal number of groups such that the part types of a group can be processed without interrupting tool switches. This means that the current collection of tools in the machine is sufficient to produce all part types in the group, and the number of groups (setup occasions) is minimal.

The JGP has been studied broadly (Crama, Kolen, Oerlemans, and Spieksma, 1994; Smed, Johnsson, Puranen, Leipälä, and Nevalainen, 1999), and there exist several efficient heuristic algorithms to solve it approximately. Although the problem is NP-hard (Crama and Oerlemans, 1994), some real-life instances can be solved to optimality with 0/1-formulation and efficient integer programming software, or by constraint logic programming (Knuutila, Puranen, Johnsson, and Nevalainen, 2001). The problem itself is of practical importance, since its solution can give significant savings in high-mix-low-volume -manufacturing environments (Johnsson, 1999).

The classic JGP is no longer the only version of practical importance. Some recent flexible PCB placement machines use several feeders of different types. This lowers machinery costs and gives freedom in manufacturing. Specialized feeder types are needed due to the different sizes and packaging techniques of the components. For example, the General Surface Mount Application Machine of Universal (`www.uic.com`) supports a versatile set of alternative feeder technologies, including tape, track, multi-tube, and matrix tray feeders. Each feeder has some fixed capacity and is capable of handling a certain packaging technology, see Figure 1 for the organization of such machine. These packaging technologies define the type of each feeder.

The present paper studies the above extension of the JGP with *multiple feeders of several types* (JGP-T). As in the common JGP, *pre-emption of the processing is not allowed* and *the order of processing the groups has no effect on the production costs*. The latter assumption

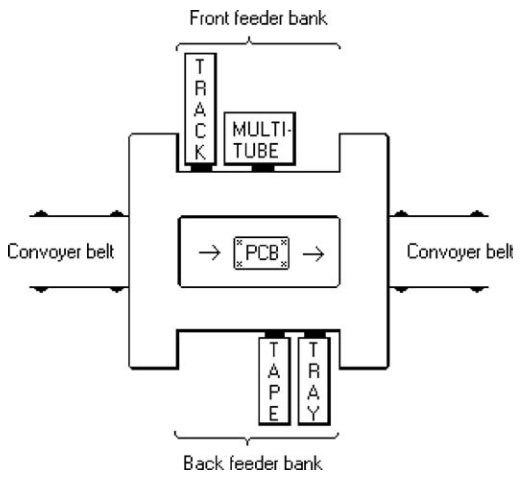


Figure 1. A possible configuration of four component and feeder types.

is valid when all feeders are changed or rearranged at once (using changeable feeder banks), or the change time is small in comparison to the fixed costs.

We start by giving a mathematical formulation of the problem in Section 2. In Sections 3 and 4 we propose a family of heuristics for JGP-T. Section 5 contains results of practical tests with these algorithms. The paper is closed by concluding remarks in Section 6.

2. Mathematical formulation of JGP-T

The formal definition of JGP-T and its integer programming formulation follows the ideas and terminology of Tang and Denardo (1988) whenever suitable.

2.1. *Definition of JGP-T*

A JGP-T problem has the following concepts:

- There are $n_{\text{feeders}} \in \mathbb{N}$ different *feeder types*, and a fixed *capacity* $c_f \in \mathbb{N}$ associated with each $f = 1, \ldots, n_{\text{feeders}}$. The types express the kind of technology used in the feeding of the electronic components, and the capacities correspond to the total number of feeder slots in the feeders for each such technology.
- There are $n_{\text{comp}} \in \mathbb{N}$ different *component types*, with an associated feeder type $f(i) \in \{1, \ldots, n_{\text{feeders}}\}$ and *capacity requirement* $r(i) \in \mathbb{N}$ for each $i = 1, \ldots, n_{\text{comp}}$. The numbers $r(i)$ stand for the amount of capacity (number of feeder slots, area in a tray) the components require in their feeder. This requirement is extended to sets of component types in the natural way, and the *feeder type—restricted capacity requirement* $r(T) \mid f\ T \subseteq \{1, \ldots, n_{\text{comp}}\}$ is defined as

$$r(T) \mid f = \sum_{i \in T, f(i)=f} r(i).$$

- There is a set $J = \{1, \ldots, n_{\text{parts}}\}$ of *part types* (e.g. PCB types). We denote with T_j the set of all component types of part j.
- A *group* is any set of part types. A group is *feasible* if no capacity constraint is violated for any feeder type. Let $S \subseteq J$. Then $T(S)$ denotes the set of components of the parts in S, that is $T(S) = \bigcup_{j \in S} T_j$. Now, the feasibility of groups can be stated formally as

$$r(T(S)) \mid f \leq c_f \quad \text{for all } f = 1, \ldots, n_{\text{feeders}}.$$

- A *grouping* is any set of groups. Grouping $\mathbb{S}$ of J is *feasible*, if $\mathbb{S}$ is a partition of J) and each member of $\mathbb{S}$ is feasible.

The *job grouping problem with several feeder types* (JGP-T) is to find a feasible grouping of J with a minimum number of groups. In comparison to the JGP we have introduced the feeder types $f = 1, \ldots, n_{\text{feeders}}$, capacities c_f for each feeder type f, and the feeder types $f(i)$ for each component type i. Other concepts have been preserved the same.

2.2. JGP-T as an integer programming problem

JGP has been formulated as a mathematical optimization problem in many works (Nemhauser and Wolsey, 1988; Crama, Oerlemans, and Spieksma, 1994). Our formulation of JGP-T is an extension of the one in Knuutila et al. (2001). We use below the abbreviation 'iff' for 'if and only if'.

Let n_{groups} be an upper bound for the number of groups in the solution. Now, if the problem is solvable then $n_{\text{groups}} \le n_{\text{parts}}$. Since each $j \in J$ must be placed in some $S_k \in \mathbb{S}$ ($1 \le k \le n_{\text{groups}}$), we denote the group containing j by a decision variable x_{jk}, where

$$x_{jk} = 1 \quad \text{iff } j \in S_k \quad \text{for all } j = 1, \dots, n_{\text{parts}}, \quad k = 1, \dots, n_{\text{groups}}.$$

The contents of $T(S_k)$ are expressed with an n_{comp}-ary vector u_k, where

$$u_{ki} = 1 \quad \text{iff } i \in T(S_k) \quad \text{for all } i = 1, \dots, n_{\text{comp}}.$$

Since our aim is to minimize the number of groups, we introduce a 0/1-decision variable s_k for each tentative group S_k, such that s_k is 1 iff S_k is used in the grouping, i.e. $S_k \neq \emptyset$.

Recall the function $f(i)$, which returns the feeder type in which components of type i can be stored. The restrictions on the decision variables are now:

$$\begin{aligned}
\sum_{i:f(i)=f} u_{ki} * r(i) &\le s_k * c_f && \text{for } k = 1, \dots, n_{\text{groups}}, \quad f = 1, \dots, n_{\text{feeders}} && (1)\\
x_{jk} &\le u_{ki} && \text{for all } i \in T_j,\ j = 1, \dots, n_{\text{parts}}, \quad k = 1, \dots, n_{\text{groups}} && (2)\\
\sum_{k=1}^{n_{\text{groups}}} x_{jk} &= 1 && \text{for } j = 1, \dots, n_{\text{parts}} && (3)\\
u_{ki} &\le s_k && \text{for } k = 1, \dots, n_{\text{groups}}, \quad i = 1, \dots, n_{\text{comp}} && (4)\\
u_{ki} &\in \{0, 1\} && \text{for } k = 1, \dots, n_{\text{groups}}, \quad i = 1, \dots, n_{\text{comp}} &&\\
x_{jk} &\in \{0, 1\} && \text{for } j = 1, \dots, n_{\text{parts}}, \quad k = 1, \dots, n_{\text{groups}} &&\\
s_k &\in \{0, 1\} && \text{for } j = 1, \dots, n_{\text{parts}}, \quad k = 1, \dots, n_{\text{groups}} && (5)
\end{aligned}$$

Constraint (1) is for the capacity restrictions (2) ensures that part j can be placed in group k only if all of it's components are there (3) causes each part to appear in exactly one group, and (4) allocates a group if we place any components into it.

Program JGP-T. Minimize

$$\sum_{k=1}^{n_{\text{groups}}} s_k$$

subject to (1)–(5).

Since the above formulation may have many equivalent solutions (e.g., permutations), it is useful to add auxiliary restrictions ruling out at least some of these. First, the groups are allocated in the lexical order:

$$s_k \geq s_{k+1} \quad \text{for all } k = 1, \ldots, n_{\text{groups}} - 1. \tag{6}$$

Secondly, we demand that the groups are arranged in a descending order (according to their size).

$$\sum_{j=1}^{n_{\text{parts}}} x_{j,k+1} \leq \sum_{j=1}^{n_{\text{parts}}} x_{jk} \quad \text{for } k = 1, \ldots, n_{\text{groups}} - 1. \tag{7}$$

Observe that we have avoided using the component-part matrix which appears commonly in the formulations of the standard JGP. The role of this matrix is now taken by the sets T_j in constraint (2). Furthermore, we could use a weaker constraint $x_{jk} \leq s_k$ (for $k = 1, \ldots, n_{\text{groups}}$) for the placement of parts into groups, but this is dominated by constraint (4). Finally note that if we fix the grouping size and just test whether a grouping of given size exists, we can simplify the model by eliminating variables s_k from it.

2.3. *Relation of JGP-T to known optimization problems*

One can consider JGP-T as a *clustering problem* (Jain and Dubes, 1988; Mirkin, 1996), where the part groups correspond to clusters and the purpose is to find the minimal number of clusters fulfilling the inclusion condition for the components in each part. Although this condition is strange for most clustering situations, we think that many similarity measures applied in efficient clustering algorithms are also applicable in grouping problems.

JGP-T is also related to the *capacitated facility location problem* (CFLP) (Aikens, 1985; Mirchandani and Francis, 1990; Chudak and Shmoys, 1999). Here we can interpret the feeders as capacitated facilities which must serve the parts (sites). As a difference to the CFLP we have multiple request demands (components) in each site and also the service types at the facilities are of different types (feeder types). In addition, distances between the facilities and sites do not play a role in JGP-T, unlike in CFLP. When comparing the mathematical models of the two problems, especially constraints 2 and 3 are similar. This lets us to believe that optimization techniques (Reeves, 1995; Papadimitriou and Steiglitz, 1998) used for solving hard combinatorial problems (like clustering, set covering, partitioning, bin packing etc.) are strong candidates for the solution of JGP-T, too.

Interpretation of JGP-T as a generalization of JGP is immediate: a JGP model has only one feeder and thus only one feeder type. This gives us many benefits. First, the solution techniques proposed for JGP are good candidates for building blocks for JGP-T techniques. Secondly, complexity bounds for JGP can be utilized when analyzing JGP-T. Especially, it has been shown (Crama and van de Klundert, 1999) that JGP is a generalization of the set covering problem and thus NP-hard. We therefore know also that *JGP-T is NP-hard*. Thirdly, following from the properties of the set covering problem, approximation of JGP even with a factor of $d \log n_{\text{parts}}$ ($d < 1/4$) is extremely improbable (cf. Theorem 7 in Crama and van de Klundert, 1999), and therefore JGP-T must be at least as hard to approximate. Nevertheless, exact solutions of JGP can often be found for problems of practical size using mathematical programming packages (Knuutila et al., 2001). Moreover, the most efficient heuristics give nearly optimal results for practical problems. Thus, by relaxing JGP-T to JGP we get lower bounds for JGP-T in cases where the exact solving is impossible.

3. Local search heuristics for JGP-T

Since the heuristics proposed in this work are based on our JGP algorithms (Smed et al., 1999; Knuutila et al., 2001), we start by recalling them briefly.

3.1. *JGP algorithms*

The JGP heuristic begins by creating a singleton grouping $\mathbb{S}_0 = \{\{j\} \mid j = 1, \ldots, n_{\text{parts}}\}$ and it then greedily merges groups according to some *similarity metric* of groups. This is iterated as long as the grouping remains legal.

The *similarity of two component sets*, *sim*, is defined as the accumulated capacity requirement of their common components:

$$sim(T_1, T_2) = \sum_{i \in T_1 \cap T_2} r(i), \quad (T_1, T_2 \subseteq \{1, \ldots, n_{\text{comp}}\}),$$

and the *similarity of groups* is $sim(S_1, S_2) = sim(T(S_1), T(S_2))$. We omit a detailed discussion of different similarity metrics. This is because the choice of the metric is not that crucial in the initial phase of our JGP-T algorithms: the groupings created using them will only serve as a starting point for the later steps. For more information on metrics, see (Shtub and Maimon, 1992).

Our local search algorithms for (untyped) JGP start from an initial grouping $\mathbb{S}_0$, select a pair of groups to be merged, perform the merging (which usually leads to an illegal grouping) and then attempt to *repair* the result by *moving* parts from a group to another or by *swapping* parts between two groups. In order to do this we need heuristics to choose from a given grouping $\mathbb{S}$:

- groups S_1, S_2 to be merged;
- groups S_1, S_2 and a part $j \in S_1$ such that j will be moved from S_1 to S_2; and
- groups S_1, S_2 and parts $j_1 \in S_1$ and $j_2 \in S_2$ such that j_1 and j_2 will be swapped between S_1 and S_2.

  The *merging heuristics* (called *merge rules*) select S_1 and S_2 to be merged by the following three rules. Let $S' = S_1 \cup S_2$ and the *total feeder capacity* c_{total}.

- [M1] $sim(S_1, S_2)$ is maximal;
- [M2] the capacity requirement $r(T(S'))$, is minimal; and

- [M3] *sim*(S_1, S_2) is maximal and $r(T(S')) \leq (c_{\text{total}} + b)$, where $b \geq 0$ is some small constant.

  The aim of the *move operations* is to transform an illegal grouping to feasible. Since this can not usually be done with one step, each move only tries to make the grouping 'less infeasible'. Let $S_1 \in \mathbb{S}$ be a (randomly chosen) infeasible group, $S_2 \in \mathbb{S}\backslash\{S_1\}$, $j \in S_1$, $S_1' = S_1\backslash\{j\}$ and $S_2' = S_2 \cup \{j\}$. We use three *repair rules*, i.e. heuristics for selecting the groups S_1, S_2 and part j:

- [R1] the increase in the requirement of S_2, $r(T(S_2')) - r(T(S_2))$, is minimal;
- [R2] the decrease in the requirement of S_1, $r(T(S_1)) - r(T(S_1'))$, is maximal;
- [R3] R1 and R2 are combined;

$$r(T(S_1)) - r(T(S_1')) - r(T(S_2')) + r(T(S_2)) = \max!$$

Swapping is performed by selecting $S_1, S_2 \in \mathbb{S}$ and $j_1 \in S_1$, $j_2 \in S_2$ such that j_1 and j_2 are swapped between S_1 and S_2. This is done so that the resulting grouping ($\mathbb{S}'$) has minimal requirement $\sum_{S\in\mathbb{S}'} r(T(S))$:

$$r(T(S_1)\backslash\{j_1\} \cup \{j_2\}) + r(T(S_2)\backslash\{j_2\} \cup \{j_1\}) = \min!$$

We next devise similar selection heuristics for JGP-T by re-defining the concepts of 'similarity' and 'excess' to take the feeder types into account. The problem is, that this can be done in several ways, and it is hard to predict their relative merits.

3.2. *Similarity heuristics for JGP-T*

When applying the ideas of the JGP algorithms to JGP-T, we must recast all the rules of Section 3.1 into typed format. We start by defining typed set similarity *sim*(T_1, T_2). It is then used in the selection criteria for moving and swapping; in particular we must also revise the concept of *feeder capacity excess* to consider feeder types.

Let T_1 and T_2 be two sets of components and F' the set of types occurring in them; $F' = \{f(i) \mid i \in T_1 \cup T_2\}$. We have (at least) the following choices for *sim*(T_1, T_2):

- MAX: *sim*$(T_1, T_2) = \max_{f=1,\ldots,n_{\text{feeders}}} r(T_1 \cap T_2) \mid f$, the maximal requirement of common components for one type;
- MIN: *sim*$(T_1, T_2) = \min_{f\in F'} r(T_1 \cap T_2) \mid f$, the minimal requirement of common components for one *occurring* type; and
- SUM: *sim*$(T_1, T_2) = \sum_{f=1,\ldots,n_{\text{feeders}}} r(T_1 \cap T_2) \mid f$, the requirement of all common components.

Note that

$$\max_{f\in F'} r(T_1 \cap T_2) \mid f = \max_{f=1,\ldots,n_{\text{feeders}}} r(T_1 \cap T_2) \mid f,$$

and

$$\sum_{f=1,\ldots,n_{\text{feeders}}} r(T_1 \cap T_2) \mid f = \sum_{f \in F'} r(T_1 \cap T_2) \mid f = r(T_1 \cap T_2).$$

The last case (SUM) coincides with the untyped case. MAX and MIN consider only the types with the largest and smallest common requirement, respectively. These can be interpreted as upper and lower bounds of the overall similarity. The similarities direct merging process: we hope that $S' = S_1 \cup S_2$ will be the "smaller" the more similar S_1 and S_2 are. In particular, we assume it is more probable for S' to be a feasible group, or at least it is closer to being one.

Note that we use the set F' in MIN, because otherwise the similarity of any two (even identical) sets would be 0 whenever $|F'| \neq n_{\text{feeders}}$. A similar problem occurs when defining the *average* requirement over all types: should we divide the sum with $|F'|$ or n_{feeders}? Denote these cases with AVG1 and AVG2. AVG1 may seem more reasonable, because we wouldn't like to 'punish' sets T_1 and T_2 for the reason that they both fail to contain components of some types. AVG2, however, is invariant to the distribution of components into types.

Example 1. Consider the component sets $T_1 = \{1, \ldots, 100\}$ and $T_2 = T_1$, the requirements $r(i) = 1$ for $i = 1, \ldots, 100$, and the type distributions ($n_{\text{feeders}} = 100$):

1. $f(1) = f(2) = \cdots = f(100)$ ($|F'| = 1$);
2. $f(1) = 1$, $f(2) = \cdots = f(100) = 2$ ($|F'| = 2$); and
3. $f(i) = i,\ i = 1, \ldots, 100$ ($|F'| = 100$) .

We then have the following results for different similarity functions and type distributions (in their respective order):

- MAX: $sim(T_1, T_2) = 100, 99, 1$;
- MIN: $sim(T_1, T_2) = 100, 1, 1$;
- AVG1: $sim(T_1, T_2) = 100, 50, 1$;
- AVG2: $sim(T_1, T_2) = 1, 1, 1$.

We do not commit ourselves to any particular way of computing the similarities, but define the algorithms to work with any of them. The parameter *SIM* denotes one of the values MAX, MIN, AVG1 and AVG2. For example, the similarity of two sets T_1 and T_2 is determined by calling a general function SIM(T_1, T_2, *SIM*).

Let us examine the merging rule M3 for selecting the groups in the untyped case. M3 tells us to select $S_1, S_2 \in \mathbb{S}$ such that $sim(S_1, S_2)$ is maximal and $r(T(S_1) \cup T(S_2)) \leq (c_{\text{total}} + b)$. Unfortunately, this does not work as such for the typed case, since 'c_{total}' is now specific for each type. We therefore define function EXCESS(T, *SIM*) which considers separately the excess $\max\{r(T) \mid f - c_f, 0\}$ for each type f and returns the maximal, minimal or average excess depending on the parameter *SIM*.

3.3. *Selecting two groups for merging*

Merging heuristics M1 and M2 use only the metric $sim(S_1, S_2)$, but M3 uses also the concept of excess. Similarity and excess can be defined in (at least) four different ways, it would be possible to use a different value of *SIM* for SIM(T_1, T_2, *SIM*) and EXCESS(T, *SIM*). This would lead to $4 + 4 + 16 = 24$ different ways of making the merging decision (4 ways to apply rule M1 etc.). In order to keep the situation simple we use the same similarity for both cases. This is justified by assuming that similarity and excess should 'support' each other: large similarity $sim(S_1, S_2)$ should lead to a small excess when S_1 and S_2 are merged.

Routine SELECTGROUPPAIR($\mathbb{S}$, *SIM*, *MER*) implements the heuristic selection for merging. Its arguments are the current grouping $\mathbb{S}$, the similarity *SIM*, and merging rule *MER*, and it returns the best group according to rule *MER*. We favor the pair with smaller excess in ties (for rule M3).

3.4. *Selecting a part and a group for moving*

Recall the repair rules of JGP (Section 3.1). When moving a part j from S_1 to S_2 one can minimize the increase of S_2 (R1); maximize the decrease of S_1 (R2); or minimize the overall result (R3). As before, for JGP-T, we consider minimum, maximum and average excesses over all types. All possible combinations of repair rules and typed similarities are encapsulated into function:

MOVEVALUES(S_1, S_2, j, *SIM*, *REP*)

where *REP* is one of R1, R2, and R3, see Figure 2. We give preference to moves that make the source group S_1 empty (thus decreasing the size of the grouping). Note that the types that give the largest decrease and the least increase may be (and quite often are) different when *SIM* is MIN or MAX.

Example 2. The choice of *SIM* and *REP* (12 possibilities) has a notable effect on the repair process. Consider the bad choice R1, MIN. Then j in S_2 is determined by the type f for

```
function MOVEVALUE($S_1, S_2, j, SIM, REP$): Real
    decrease := EXCESS($T(S_1)$, SIM) - EXCESS($T(S_1 \setminus \{j\})$, SIM)
    increase := EXCESS($T(S_2 \cup \{j\})$, SIM) - EXCESS($T(S_2)$, SIM)
    case REP of
        R1: value := increase
        R2: value := decrease
        R3: value := decrease - increase
    if $|T(S_1)| = 1$ then return $\infty$
    else return value
end MOVEVALUE
```

Figure 2. Function computing the heuristic value of a move operation.

which $r(T(S_2 \cup \{c\})) \mid f - c_f$ is minimal (MIN). Rule R1 selects the group-part -pair with the smallest such value. Suppose that

- $j = \{1, 2, \ldots, 100\}$, $f(1) = 1$, and $f(2) = \cdots = f(100) = 2$;
- $j' = \{2, 101\}$, where $f(101) = 3$;
- $T(S_2) = \{3, 102\}$, where $f(102) = 3$;
- $c_1 = c_2 = c_3 = 1$.

If $r(i) = 1$ for all $i = 1, \ldots, 102$ we get

$$\begin{aligned}
&\text{EXCESS}(T(S_2), \text{MIN}) = \min\{1-1, 1-1\} = 0,\\
&\text{MOVEVALUE}(S_1, S_2, j, \text{MIN}, \text{R1}) = \min\{1-1, 100-1, 1-1\} - 0 = 0,\\
&\text{MOVEVALUE}(S_1, S_2, j', \text{MIN}, \text{R1}) = \min\{2-1, 2-1\} - 0 = 1,
\end{aligned}$$

and we would prefer j to j'. However, the groups resulting from the corresponding moves have overall excesses of 99 (for j) and 2 (for j'). Thus, using R1 and MIN leads to a very misleading estimator for the repair value.

Figure 3 contains the implementation of routine SELECTMOVE which returns the best triple S_1, S_2, j (according to *SIM* and *REP*). Note that the selection of S_1 is restricted on illegal groups. Since R2 measures only the value of S_1, the choice of S_2 can be arbitrary when *REP* = R2.

```
function SelectMove($\mathbb{S}$, SIM, REP): group, group, part
    if $R$ = R1 then bestval := $\infty$ else bestval := $-\infty$
    for each $S_1 \in \mathbb{S}$ do
        if $r(S_1)|f > c_f$ for some $f = 1, ..., n_{feeders}$ then
            $\mathbb{S}' := \mathbb{S} \setminus \{S_1\}$
            if REP = R2 then
                draw a random $S_2$ from $\mathbb{S}'$; $\mathbb{S}' := \{S_2\}$
            for each $S_2 \in \mathbb{S}'$ do
                for each $j \in S_1$ do
                    update := False
                    val := MoveValue($S_1, S_2, j$, SIM, REP)
                    if $R$ = R1 then update := (val < bestval)
                    else update := (val > bestval)
                    if update then
                        $bestj := j$; $Best_1 := S_1$; $Best_2 := S_2$;
                        bestval := val
    return $Best_1, Best_2, bestj$
end SelectMove
```

Figure 3. Selecting the best move operation.

3.5. Swapping of parts

As before, we must analyze the capacity excesses for each type. Routine

SELECTSWAP($\mathbb{S}$, *SIM*)

selects the objects of the swap operation (2 groups and 2 parts) by iterating over all alternatives and selecting the best local improvement, i.e. the swap maximizing the difference between the current excess and the excess after the swap.

4. Algorithms for JGP-T

One can combine the merge-, move- and swap-operations of Section 3 in many alternative ways to form up a working JGP-T algorithm. We examine four such variants in the order of increasing complexity.

4.1. Greedy clustering algorithm

The main use of the clustering algorithm

GREEDY(*SIM*): grouping

is to produce an initial solution for the improvement algorithms. It also serves as a comparison point for other methods. GREEDY starts from an initial grouping of singular groups (one for each part). Group pairs with maximal SIM(S_1, S_2, *SIM*) value are then merged as long as the capacity constraints are not violated. Note that we do not use SELECTGROUPPAIR in the selection, since GREEDY is designed to work without any merge rule and also because it considers only mergings that preserve the feasibility.

4.2. Local and global search

Algorithm

LOCALJGP-T($\mathbb{S}$, *SIM*, *MER*, *REP*, *maxIter*): grouping

starts from an initial grouping (e.g. one given by GREEDY) which it tries to improve further. If the grouping becomes illegal, we attempt to repair by using move and swap operations. There are many possibilities to select the order of the operations, we use a randomized strategy:

- decide whether to move or swap by a probabilistic coin toss;
- determine the locally best operands using either SELECTMOVE or SELECTSWAP and execute the operation;
- bound the amount of allowed repair operations by a constant (to guarantee termination).

Note that we do not check whether the selected operation actually decreases the amount of excess in the grouping. This is because the selection routines themselves aim to return such decreasing operations, and return ‘worse’ ones only when there’s no alternative.

If the repairing was successful (or the grouping was feasible in the first place), the current grouping is decreased by one group by merging the two groups returned by SELECTGROUPPAIR, after which the repair process is repeated. In an unsuccessful case we return the last legal grouping. Thus, if the initial grouping (given to LOCALJGP-T) was illegal and the algorithm fails to repair it, the grouping remains illegal.

Algorithm

GLOBALJGP-T($\mathbb{S}$, *SIM*, *MER*, *REP*, *timeLimit*): grouping

makes several local searches using LOCALJGP-T. The first search is done using the result of GREEDY as the starting point; the later ones are done from randomly created (possibly illegal) groupings. Searches are performed until a given time limit is exceeded.

4.3. Tabu search algorithm

In order to implement a tabu search (TS) algorithm for JGP-T, we need a *neighborhood function*, an *improvement function* and an *aspiration criterium* for groupings. The neighborhood of a grouping is defined as the set of groupings which can be reached from the current one by swapping a part pair between two groups or by moving a single part from a group to another. These two operations also form the contents of the tabu list during the search. The aspiration criterium allows a tabu operation to be used if it would decrease the number of groups in the current grouping. The tabu search algorithm.

TS($\mathbb{S}$, *SIM*, *REP*, *Tenure*, *Limit*): grouping

is similar to LOCALJGP-T with the following differences:

- groups are not explicitly merged but groupings get smaller as a consequence of move operations; and
- current grouping is always feasible.

5. Computational results

The goals of our testing were

- to find out which choices for *SIM*, *MER* and *REP* work best for JGP-T;
- to compare Local, Global and Tabu searches using the most promising parameter combinations from the previous test; and
- to compare the results to optimal solutions (when possible).

Our test data is based on a real-life case consisting of 157 electronic components, 41 PCB types and 4 different feeder types. The capacity requirements of the components are between 3 and 7 units, and the feeder capacities vary between 100 and 500. Tests were executed on randomly drawn PCB type samples of size 25. Almost all of our algorithms performed equally well for these cases. In order to discover differences between the alternatives we therefore generated a larger synthetic test case of 200 PCB types as follows. First, the component sets of the 41 real PCB types were randomly divided into two halves (yielding 82 sets). We then drew random pairs from these halves (from the population of size $82 \times 81 = 7452$) and synthesized a new PCB as the union of these pairs. Each new PCB was checked for duplicates and the process was continued until we had 200 different PCB types in total. Tests were then executed by taking samples of 100 PCB types from this set of 200. Both the Global and Tabu search used a time limit of 60 seconds, and the tabu tenure was 30. The maximum number of iterations in the Local was 1000.

We used ILOG Solver and our integer programming formulation to see whether the solutions generated by our heuristics were optimal. This was done by verifying that the problem had no solution when the grouping was constrained to be one group smaller. It turned out that the running times of ILOG were impractically large (taking several days to solve) for samples of 30 PCB types and up, while (untyped) JGP instances of 50 PCB types are normally easy to solve optimally (Knuutila et al., 2001). However, all the problems of size 25 were solved reasonably fast (all but one of our test cases were solved in less than 15 minutes).

5.1. *Finding out candidates for SIM, MER and REP*

The first series of tests was performed to filter out the most and least promising candidate combinations for *SIM*, *MER*, and *REP* to be examined further. The test data consisted of 30 samples of size $n_{\text{parts}} = 100$ from the population of 200 synthesized PCBs. We tried all the different combinations of heuristic algorithms, and computed the average grouping sizes for each value of *SIM* (MIN, MAX, AVG1, AVG2), *MER* (M1, M2, M3), and *REP* (R1, R2, R3). These averages were obtained by summing up all the results from combinations having a particular value for *SIM* (9), *MER* (12), and *REP* (12). By this kind of averaging we can for example find a *SIM*-choice which works well for the possible *MER*- and *REP*-choices in general. We omit here data for all possible (*SIM*, *MER*, *REP*)-combinations.

Table 1 contains average grouping sizes from our test runs. The rows for *MER* and *REP* are empty for the greedy algorithm, because its results are affected by the similarity rule only. For the same reason, rows for *MER* are empty for TS. The results hint that that rule MAX could be the best value for *SIM* (and MIN the worst). A closer examination supports this observation strongly. First, when all the 36 combinations were sorted according to the average group sizes, value $SIM =$ MIN took the last 9 positions and value $MIN =$ MAX took 9 out of the 10 first positions. Secondly, pairwise t-test of MAX against MIN, AVG1, and AVG2 revealed that the difference is statistically significant with a confidence level of over 99.5% for all cases.

The selection of *MER* and *REP* does not seem to have much effect to the results. Although M1 and R2 give slightly better average results than the other values, the differences are not statistically significant.

Table 1. Results of the 30 test runs for samples of 100 PCB types. The numbers indicate the average grouping size, where averages are computed over different parameter combinations; 9 cases for *SIM*, and 12 cases for *MER* and *REP*.

	Average $\lvert\mathbb{S}\rvert$			
	Greedy	Local	Global	Tabu
SIM				
MIN	7.67	7.60	7.60	5.36
MAX	4.55	4.54	4.54	4.50
AVG1	4.77	4.77	4.77	4.59
AVG2	5.13	5.13	5.13	4.74
MER				
M1	–	5.41	5.41	–
M2	–	5.43	5.43	–
M3	–	5.44	5.44	–
REP				
R1	–	5.43	5.43	4.77
R2	–	5.41	5.40	4.77
R3	–	5.44	5.44	4.78

5.2. *Comparision of search algorithms*

The best individual combinations of the ones tested in the previous section were MAX/M3/R1 and MAX/M3/R3 (average grouping size 4.50), while combinations MAX/M2/R1, MAX/M2/R2 and MAX/M3/R2 were only slightly worse. The results of Table 1 suggest that no matter what the values are, TS gives the smallest groupings. Table 2 contains the results for these 5 parameter combinations. TS is still best in all combinations, but the differences are not statistically significant. The larger differences with other parameter

Table 2. Results of different algorithms using the 5 best parameter combinations (30 test runs for samples of 100 PCB types).

	Average $\lvert\mathbb{S}\rvert$			
Parameters	Greedy	Local	Global	Tabu
MAX/M3/R1	4.53	4.50	4.50	4.47
MAX/M3/R3	4.53	4.50	4.50	4.47
MAX/M2/R1	4.53	4.53	4.53	4.47
MAX/M2/R2	4.57	4.53	4.53	4.47
MAX/M3/R2	4.53	4.53	4.53	4.50

Table 3. Results of different algorithms using 'good' (MAX/M3/R1) and 'bad' (MIN/M1/R1) parameter combinations, compared against optimal solutions (30 test runs for samples of 25 PCB types).

	Average $\|\mathbb{S}\|$	
Method	MAX/M3/R1	MIN/M1/R1
Greedy	3.03	3.53
Local	3.03	3.53
Global	3.03	3.37
Tabu	3.03	3.20
MIP (ILOG)	3.00	3.00

values (especially when *SIM* is MIN) show that TS is least sensitive to the quality of the initial grouping provided by Greedy algorithm. One can also observe that the results for Local and Global searches are identical, and that Greedy was only slightly inferior to Local (and Global). Actually, Global was able to improve the grouping only in 2 cases out of the 1080 tested. This is quite different from the results with JGP, where the difference was statistically significant (Smed et al., 1999; Knuutila et al., 2001).

5.3. Comparision to optimal solutions

We ran similar tests for smaller problems of 25 real PCBs sampled from the population of 41 in order to find out how close to the optimum the best methods are able to get. For problems of this size, all the proposed algorithms found an optimal solution for 29 out of the 30 test cases (when the best parameter values were used). Tabu search got quite close to optimum even with bad parameter combinations. The results of Table 3 show that the parameters have a remarkable effect on the quality of the result of all search methods, and also that our methods produce almost optimal results for this problem set.

6. Concluding remarks

The job grouping problem in the case of several feeder types was discussed. The problem is a natural generalization of the well-known standard JGP. Since modern PCB assembly production uses multiple different packaging techniques, the types of feeders and electronic components should be included also in the mathematical model of JGP. While the introduction of the feeder/component types seems to cause only moderate changes in the problem description, the effect on the solution algorithms turned out to be significant. The reason for this lies in the difficulties of measuring the similarity of two PCB types when searching for advantageous local modifications of the present grouping.

JGP-T resembles several known hard combinatorial optimization problems (e.g., clustering and capacitated facility location). It is thus no coincidence that our solving efforts use similar local and global optimization techniques as have been used with these problems. In particular, we constructed four solution algorithms (Greedy, Local search, Global search and Tabu search) and used parameterized options for defining the similarity, merging rules and repair rules. The tests of Section 5 were performed to analyze the role of these parameters and the importance of using more complex heuristics instead of simple greedy solution. We used rather large problem instances in the tests in order to make the differences between the various parameter settings more clear. According to our previous experience with JGP one may then get more variation in the results. A drawback of the large problem sizes was our unability to solve the problems to optimality. It is however interesting to know that when considering small problem instances ($|\mathbb{S}| \leq 4$ in an optimal solution) with 25 PCB types, almost all of our heuristic algorithms also found an optimal solution. Tabu search gave the overall best results while the Local abd Global search algorithms and the Greedy algorithm worked equally well in practice. In the light of related research, we expect that evolutionary algorithms were also a good candidate for the JGP-T.

As future directions for this line of research we note that a simplifying assumption was made that each component has a unique feeder type. In practice, it is possible that the same component can be placed in the feeder unit using different feeder types. Another modification to the basic setting is the availability of flexible feeders (so called feeder boxes) (Hirvikorpi, Knuutila, Johnsson, and Nevalainen, 2003). Despite the fact that we are only taking the first faltering steps in this research, many ideas and algorithms have been already implemented in the Trilogy5000 engineering system for assembly and test of PCB boards of Valor Ltd (`www.valor.com`).

Acknowledgments

The authors wish to thank the Associate Editor and the anynymous referees for their exceptionally constructive and useful comments on our manuscript. Among others, the associate editor pointed us a simplified mathematical formulation of the problem and connections of various combinatorial problems to JGP-T.

References

Aikens, C. H., "Facility Location Models for Distribution Planning," *European Journal of Operational Research*, Vol. 22, pp. 263–279 (1985).

Chudak, F. A. and Shmoys, D. P., "Improved Approximation Algorithms for a Capacitated Facility Location Problem," *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 875–876 (1999).

Crama, Y., Kolen, A. W. J., Oerlemans, A. G., and Spieksma, F. C. R., "Minimizing the Number of Tool Switches on a Flexible Machine," *International Journal of Flexible Manufacturing Systems*, Vol. 6, pp. 33–54 (1994).

Crama, Y. and Oerlemans, A., "A Column Generation Approach to Job Grouping for Flexible Manufacturing Systems," *European Journal of Operational Research*, Vol. 78, pp. 58–80 (1994).

Crama, Y., Oerlemans, A., and Spieksma, F., *Production Planning in Automated Manufacturing*, Vol. 414 of *Lecture Notes in Economics and Mathematical Systems*, Springer-Verlag, Germany (1994).

Crama, Y. and van de Klundert, J., "Worst-case Performance of Approximation Algorithms for Tool Management Problems," *Naval Research Logistics*, Vol. 46, pp. 445–462 (1999).

Hirvikorpi, M., Knuutila, T., Johnsson, M., and Nevalainen, O., "Grouping of PCB Assembly Jobs in the Case of Flexible Feeder Units," *Proceedings of Group Technology/Cellular Manufacturing World Symposium*, Columbus, Ohio, pp. 195–201 (2003).

Jain, A. and Dubes, R., *Algorithms for Clustering Data*, Prentice-Hall, Englewoods Cliffs, New Jersey (1988).

Johnsson, M., “Operational and Tactical Level Optimization in Printed Circuit Board Assembly,” PhD thesis, University of Turku, Finland (1999).

Knuutila, T., Puranen, M., Johnsson, M., and Nevalainen, O., “Three Perspectives for Solving the Job Grouping Problem,” *International Journal of Production Research*, Vol. 39, No. 18, pp. 4261–4280 (2001).

Mirchandani, P. and Francis, R., *Discrete Location Theory*, John Wiley and Sons, New York (1990).

Mirkin, B., *Mathematical Classification and Clustering (Nonconvex Optimization and Its Applications)*, Kluwer Academic Publishers, Dordrecht, The Netherlands (1996).

Nemhauser, G. L. and Wolsey, L. A., *Integer and Combinatorial Optimization*, Wiley (1988).

Papadimitriou, C. H. and Steiglitz, K., *Combinatorial Optimization: Algorithms and Complexity*, 2nd edn, Dover Publications, Mineola, New York (1998).

Reeves, C. R., *Modern Heuristic Techniques for Combinatorial Problems*, McGraw-Hill (1995).

Shtub, A. and Maimon, O., “Role of Similarity in PCB Grouping Procedures,” *International Journal of Production Research*, Vol. 30, No. 5, pp. 973–983 (1992).

Smed, J., Johnsson, M., Puranen, M., Leipälä, T., and Nevalainen, O., “Job Grouping in Surface-Mounted Component Printing,” *Robotics and Computer-Integrated Manufacturing*, Vol. 15, No. 1, pp. 39–49 (1999).

Tang, C. S. and Denardo, E. V., “Models Arising from a Flexible Manufacturing Machine,” *Operations Research*, Vol. 36, No. 5, pp. 767–784 (1988).

Publication II

Knuutila T., Hirvikorpi M., Johnsson M. and Nevalainen O.S., Grouping of PCB Assembly Jobs in the Case of Flexible Feeder Units. *Engineering Optimization*, 37(1), 2005.

Engineering Optimization
Vol. 37, No. 1, January 2005, 29–48

Grouping of PCB assembly jobs in the case of flexible feeder units

TIMO KNUUTILA†*, MIKA HIRVIKORPI†, MIKA JOHNSSON‡
and OLLI S. NEVALAINEN†

†Turku Centre for Computer Science, University of Turku, 20014 Turku, Finland
‡Valor Computerized Systems (Finland), Ruukinkatu 2, 20540 Turku, Finland

(Received 30 September 2003; in final form 20 April 2004)

Change costs between jobs in printed circuit board assembly depend on the number of component change occasions. Component changes are necessary due to the restricted feeder capacity. Grouping assembly jobs decreases the number of setup occasions. The job grouping problem asks for a minimal cardinality grouping.

This paper studies a generalization of the job grouping problem where the feeder unit is capable of holding sub-units, called feeder boxes. There is a limited set of boxes available in an auxiliary storage, and the feeder unit has certain limitations for the feeder boxes it may hold. The task in job grouping problem with boxes (JGP-B) is to group the jobs into a minimal number of groups in such a way that capacity and feeder restrictions are not violated. Exact solving of the JGP-B is restricted to small problem instances only. Several heuristics are proposed and their operation is tested experimentally.

Keywords: Electronics assembly; Single machine job grouping; Heuristics; Combinatorial optimization

1. Introduction

Manufacturing processes of printed circuit board (PCB) assembly offer several different alternatives to increase production efficiency. Minimizing the time factors of the various job phases often gives a significant speedup and research on the methods of production control has therefore gained substantial interest, see Tang and Denardo (1988), McGinnis *et al.* (1992), Crama (1997), Crama *et al.* (2002), and Smed (2002) for some literature on the subject. One fruitful track in this research has focused on the minimization of the number of job change occasions. Switching from the production of a batch of PCBs of a particular type (often called a job or a PCB job) to another on an automated component placement machine requires several time-consuming operations. Some of these operations demand a fixed time, like changing the numeric control program or supplying a new batch of bare PCBs to the machine. The total cost

* Corresponding author. Email: knuutila@it.utu.fi

of these operations depends on the proper selection of batch sizes. Finding a balance between low storage levels and excessive job change costs is the key to minimize these costs.

The other type of operation times are *variable* in their nature. Although the presence of these variable cost factors depends mostly on the production machinery, the following three occur in most environments. First, the width of the conveyor belt has to be properly fitted for each PCB batch. This demands manual work taking at least a few minutes. Second, with surface mounted technology, the fixation of the electronic components is done in an oven, which should have a temperature profile correct for the current PCB. The different temperature specifications of two successive PCB types may require time consuming adjustment of the oven. The third variable cost factor is caused by component changes that originate from the fact that the necessary electronic components of the next PCB type must be present in the feeder unit prior to their placement.

The two first-mentioned variable costs are essential in many situations, but still a common approach has been to omit them (see however the fuzzy logic formulation of job grouping in Johtela *et al.* (1999)), or to handle the jobs in batches of similar urgency, width, or temperature demands. This policy is in line with the practice of showing several good solution candidates to the production engineer so the final decision can be left to a human expert and some unwanted situations, including the above-mentioned demands of certain width and temperature, could be avoided. Thus, most of the methods documented in the literature deal with minimizing the number of job groups when the feeder setup is the only criterion in their construction (see Crama *et al.* (1994) and Johnsson (1999) for references on scheduling PCB assembly jobs).

This paper considers the production control of a single placement machine in the case of the so-called high-mix low-volume production which means that the focus is in the operation of a single machine which may be a member of a production line, but is the bottleneck of the process, too. Further, the production program includes in this case many different types of relatively small PCB batches. So, the setup operation times dominate the production time and are more essential in the optimization than the time for placement itself.

Another even more essential assumption is that the feeder unit comprises a linear array of component positions which can be filled by components side by side. This kind of an abstraction is accurate enough for several surface mount devices (SMD) with a turret-like placement mechanism and a horizontally moving feeder unit (Landers *et al.* 1994). While most previous studies of the job grouping problem (JGP) use this abstraction, it is important to realize that even minor technical details of modern placement machines may degrade the accuracy of the model significantly. For example, the feeder unit may consist of removable sub-units, and the boundaries of these sub-units may not be crossed by components demanding several slots of feeder capacity. Note that this particular problem can often be solved satisfactorily by decreasing the total capacity of the feeder unit to compensate the losses on the sub-unit boundaries (Smed *et al.* 1999).

The basic JGP (with a linear feeder unit) is known to be NP-hard (Crama *et al.* 1994). Even its known approximation algorithms have very bad worst-case limits, as shown in the work of Crama and van de Klundert (1999). The cited work also contains an analysis of the approximability of tool management problems. However, the exact solution of JGP has been recently computed for several problem instances of practical size using efficient mixed integer programming (MIP) and Constraint Programming packages (ILOG Solver), see Knuutila *et al.* (2001). In addition, several very efficient heuristics have been developed for the problem (Shtub and Maimon 1992, Bhaskar and Narendran 1996, Leon and Peters 1996, Knuutila *et al.* 2001).

The job grouping problem becomes much more complicated if the assumption of a homogenous component feeder does not hold. This is the case when the machine is capable of handling components which are delivered by using different packaging technologies like feeder reels (as traditionally assumed above), tubes, trays, etc. The components have now, in addition to the type and width, the *input medium type*. Heuristics presented for this so-called job grouping problem with different component types (JGP-T) use similar ideas as for the regular JGP, see Knuutila *et al.* (2004). However, exact solutions are now unreachable for large problems due to the increased complexity, and even the heuristics become relatively involved.

This paper considers a still more general problem, called job grouping problem with feeder boxes (JGP-B). JGP-B has recently become actual due to the progress in technologies of component placement machines. As in the previous variants of JGP, this variant calls for a job grouping of minimal cardinality. The difference between JGP and JGP-B lies in the organization of the feeder unit. In order to gain extra flexibility, the unit is constructed to contain several separate *feeder boxes* that can be changed when switching jobs. The feeder unit of the machine has a certain maximal capacity (width) for storing the boxes and there is a collection (multiset) of boxes of different types to select from in an auxiliary storage.

The feeder boxes used in contemporary component placement machines are divided into *fixed* and *flexible* ones.[†] Fixed boxes are treated as a special case of flexible ones, since it is possible to store only components of a given constant width in them, and each component consumes the same amount of slots (equal to the width of the component) from the available box capacity. Flexible boxes have the extra advantage that they can hold components of different (allowed) widths. The capacity calculation is now, however, more complicated: a component may consume more than its nominal width from the capacity of a box. This excess is stated in the specifications of each box type for each allowed component width. What complicates the situation even more is that the feeder unit sets two different constraints on the feeder boxes: both the maximal number of feeder boxes and the summed up width of the boxes in the unit are restricted. Finally, the number of different available boxes is restricted simply because of financial reasons; the boxes are costly and therefore manufacturers want to acquire only a small number of boxes.

To sum up, the JGP-B calls for a minimal cardinality grouping when the set of jobs and set of boxes are given. The boxes of each job group must fit in the feeder unit, boxes must be selected from the given set, and the components of all jobs of the same group must fit in the selected boxes simultaneously. An important observation is that the basic JGP is a relaxation of the present problem.

This exposition of JGP-B is organized as follows. Section 2 defines the JGP-B formally and translates it to a MIP form. The same section also discusses the connection of JGP-B to the set-covering problem. Section 3 introduces heuristic solution algorithms for the problem. These algorithms apply several local search operations specialized for the feeder and box constraints. Section 4 contains a comparison of the proposed algorithms using grouping problems of practical size. The paper is concluded in section 5.

2. Problem statement

This section describes the JGP-B solely in terms of PCB assembly. However, JGP-B may well find its application in other flexible manufacturing environments than PCB assembly, and the interested reader is advised to consult Hirvikorpi *et al.* (2003) for a general set-theoretical problem setting of JGP-B.

[†]The feeder magazines Tape Magazine and TM-Flex Magazine of MYDATA (cf. MY-9) are examples of this kind of technology, see http://www.mydata.com/ for more information.

2.1 *Basic notation*

The following components form an instance of JGP-B.

- The set of *components* E. Each component $e \in E$ has a *width* (feeder capacity requirement) $w(e) \in \mathbb{N}$ and $w(E)$ is the set of all different widths in E.
- The set of *jobs* C. Each job (PCB type) $c \in C$ is a subset of E, *i.e.* a job is totally characterized by the set c of its components (note that this is the set of component *types* the PCB consists of, the actual numbers of occurrences of different component types are not included in this model). Each $c \in C$ and width $k \in w(E)$ together defines the subset of those components of c which have a specific width k as $E(c, k) = \{e \in c \mid w(e) = k\}$.
- The multiset of feeder boxes B. Each feeder box $b \in B$ has the following three properties:
  - $v(b) \in \mathbb{N}$, the *internal box width* (*i.e.* the capacity of the box);
  - $w(b) \in \mathbb{N}$, the *external box width* (*i.e.* the total width allocated from the feeder unit capacity) and
  - $a(b) \subseteq w(E)$, the *set of allowed component widths*.
- The feeder unit is described with two parameters:
  - the maximal total width of boxes $w_{\max} \in \mathbb{N}$ and
  - the maximal total number of boxes $n_{\max} \in \mathbb{N}$.

Based on the above notation, a box $b \in B$

- can hold a component $e \in E$ only if $w(e) \in a(b)$ (provided there is enough free space in b),
- 'box b is fixed' if $|a(b)| = 1$ and
- 'box b is flexible' if $|a(b)| > 1$.

Note also that it is reasonable to assume that

$$w(E) \subseteq \bigcup_{b \in B} a(b)$$

i.e. there is at least one box type for each component width, and also that $v(b) \leq w(b)$ (the internal width of boxes does not exceed their external width).

2.2 *Placing components into boxes*

When a component of width k is placed in b ($k \in a(b)$), it allocates at least k units of the capacity from b. In the PCB assembly it is typical that this allocation is exactly k if box b is fixed, but slightly larger than k in the flexible case. Mapping s: $B \times w(E) \to \mathbb{R}$ formalizes this property by giving each (box, component width)-pair (b, k) the capacity allocated by a component of width k in box b. This formulation also assumes that there is no slack for fixed-type boxes, *i.e.* if $a(b) = \{k\}$ then $s(b, k) = k$; and, in general, the allocation is always at least equal to the actual width of the components: $s(b, k) \geq k$. The two restrictions are by no means essential in the formulation, they just make the formulation of certain concepts (like total allocation of a component set in a box) a bit simpler.

Note that the problem of placing a set of components (of different widths) into a given multiset of boxes (of restricted widths) is clearly the well-known problem of *bin-packing* (see Martello and Toth (1990), for example), which is here further complicated by the fact that the allocations $s(b_1, k)$ and $s(b_2, k)$ may be different for some $b_1, b_2 \in B$. Similarly, it is possible that the 'absolute slack' $s(b, k) - k$ and 'relative slack' $s(b, k)/k$ are different for different component widths k (even in the same box). There is a simplifying factor too: since the slack $s(b, k)$ is minimal for the fixed boxes ($s(b, k) = k$), and it is not possible to place anything else than components of width k into these boxes, a solution algorithm to this box bin-packing problem can first use all of their capacity (and the choice of these components can be random) and thus reduce the problem size considerably.

2.3 *Groupings*

A solution of JGP-B requires the formulation of the concepts group and grouping.

- A *group* g is a subset of the set of jobs ($g \subseteq C$). The set of all components of g is denoted by $E(g)$. Each g and $k \in w(E)$ together define the set $E(g, k) = \bigcup_{c \in g} E(c, k)$ which contains all the components of width k that appear in the jobs of g.
- A *grouping* G of C is a set of groups with the property $\bigcup_{g \in G} g = C$. Because a grouping must contain all the jobs of C, there is at least one feeder setting (corresponding to a group of jobs) such that each particular job has all of its components in the feeder unit simultaneously. Observe that this does not rule out the possibility that a job may belong to several groups.

The components $E(g_i)$ of each group $g_i \in G$ should be placed in the feeder unit. This means that we must select some subset of boxes $B_i \subseteq B$ and assign each $e \in E(g_i)$ to some $b \in B_i$. Formally, each group g_i is associated with a multiset of *selected boxes* B_i and *component-box assignment* α_i: $E(g_i) \to B_i$. It is required that if $\alpha_i(e) = b$ then $w(e) \in a(b)$, *i.e.* only components of allowed widths can be placed into any box.

2.4 *Feasibility*

An important concept in the formulation of JGP-B is the ‘feasibility of grouping G’. This concept, however, requires the definitions of ‘feasibility of B_i’ and the ‘feasibility of group g_i’.

- A multiset of boxes $B_i \subseteq B$ is feasible with respect to $n_{\max}$ and $w_{\max}$ if $|B_i| \le n_{\max}$ and

$$w(B_i) = \sum_{b \in B_i} w(b) \le w_{\max}$$

  This definition says that the number of the feeder boxes and their total width should fit the constraints of the feeder unit.
- Group g_i is feasible with respect to B_i, if the boxes of B_i can hold all the components of g_i. More formally, there must exist a totally defined component-box assignment α_i: $E(g_i) \to B_i$ such that

$$\sum_{e \in \alpha_i^{-1}(b)} s(b, w(e)) \le v(b) \quad \text{for all } b \in B_i$$

Note that in practice the space allocation is not always commutative, *i.e.* different permutations of components may require different amounts of capacity. The standard way around this difficulty is to first find the groupings and then decide on the ordering of the components in the group.

A grouping G is feasible with respect to a box multiset B and constants $w_{\max}, n_{\max} \in \mathbb{N}$ if there is for each $g_i \in G$ a multiset of selected boxes $B_i \subseteq B$ such that g_i is feasible with respect to B_i and B_i is feasible with respect to $w_{\max}$ and $n_{\max}$. Thus, a total characterization of a feasible grouping G consists of a set of (feasible) triplets (g_i, B_i, α_i), $i = 1, \ldots, |G|$.

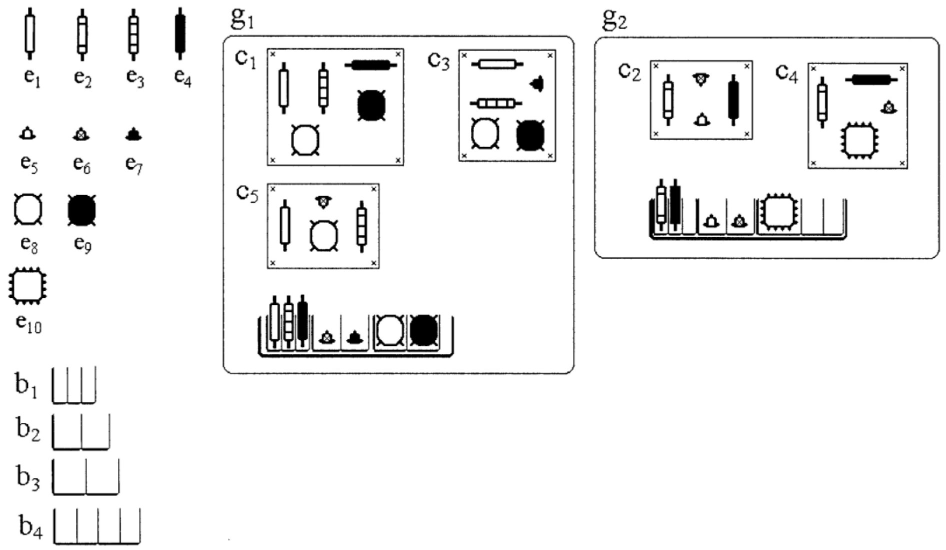


Figure 1. A JGP-B problem ($E = \{e_1, \ldots, e_{10}\}$, $C = \{c_1, \ldots, c_5\}$, $B = \{b_1, \ldots, b_4\}$) and a grouping of size 2.

2.5 *A JGP-B example*

Figure 1 contains an illustration of a JGP-B problem. There is one flexible box (b_4) and three fixed boxes (b_1, b_2 and b_3). The width of the feeder unit is 18 and it can hold up to four boxes simultaneously. The optimal solution is also shown in figure 1.

2.6 *Mixed integer programming formulation of JGP-B*

This section formulates the JGP-B. An instance of JGP-B:

- E, the set of components;
- C, the set of jobs;
- B, the multiset of boxes;
- $w_{\max}$, the maximal total width of boxes in the feeder unit (bound on $w(B_i)$) and
- $n_{\max}$, the maximal allowed number of boxes in the feeder unit (bound on $|B_i|$),

asks for a feasible grouping G with minimal cardinality.

The JGP-B is much more complicated to solve than the basic JGP. This is due to the fact that in addition to a minimal-size grouping, it asks for each group a set of boxes and a mapping of components to these boxes. The difficulty here is that even solving the feasibility of a group against a box set is a generalization of the bin-packing problem (which is known to be hard). The difficulty of the feasibility checking comes from the fact that the order of placing the components into boxes has an effect on the result. Note further that, theoretically, searching for all feasible box multisets from the multiset of all boxes is a hard problem. However, the small number and relatively large width of the boxes makes this problem easier in practice.

Our JGP-B formulation translates itself to a constraint satisfaction problem which is solvable by modern optimization packages (*e.g.* ILOG Solver). The translation is as follows:

$W = \max\{w(e) \mid e \in E\}$;
$K =$ maximum number of groups ($\leq |C|$ if the problem is solvable);

$e_{iq} = 1$ if the width of component i is q, 0 otherwise ($i = 1, \dots, |E|, q = 1, \dots, W$) and $A_{ij} = 1$ if component i is required by job j, 0 otherwise ($i = 1, \dots, |E|, j = 1, \dots, |C|$).

The decision variables are:

$x_{jk} = 1$ if job j is in group k, 0 otherwise ($j = 1, \dots, |C|, k = 1, \dots, K$);
$y_{bk} = 1$ if box b is in the configuration of group k, 0 otherwise ($b = 1, \dots, |B|, k = 1, \dots, K$) and
$z_{ibk} = 1$ if component i is in box b of group k, 0 otherwise, ($i = 1, \dots, |E|, b = 1, \dots, |B|, k = 1, \dots, K$).

A solution of the JGP-B must satisfy the following constraints:

$$\sum_{k=1,\dots,K} x_{jk} \geq 1 \quad \text{for all } j = 1, \dots, |C| \tag{1}$$

$$\sum_{i=1,\dots,|E|} \sum_{q=1,\dots,W} e_{iq} z_{ibk} s(b, q) \leq v(b) \quad \text{for all } b = 1, \dots, |B| \text{ and } k = 1, \dots, K \tag{2}$$

$$\sum_{b=1,\dots,|B|} z_{ibk} \geq x_{jk} A_{ij} \quad \text{for all } i = 1, \dots, |E|, j = 1, \dots, |C| \text{ and } k = 1, \dots, K \tag{3}$$

$$\sum_{b=1,\dots,|B|} y_{bk} w(b) \leq w_{\max} \quad \text{for all } k = 1, \dots, K \tag{4}$$

$$z_{ibk} \leq y_{bk} \quad \text{for all } i = 1, \dots, |E|, b = 1, \dots, |B| \text{ and } k = 1, \dots, K \tag{5}$$

$$x_{jk}, y_{bk}, z_{ibk} \in \{0, 1\} \quad \text{for } b = 1, \dots, |B|, i = 1, \dots, |E|, j = 1, \dots, |C| \text{ and } k = 1, \dots, K \tag{6}$$

In this formulation there is a fixed number of groups K and the minimal K is found by repeated solution trials of the satisfaction problem (1)–(6). Constraint (1) says that every job must be in at least one group. Constraint (2) demands that no capacities are exceeded in any box. Constraint (3) forces the components which are needed by a job to be in the feeder. Constraint (4) says that the boxes must fit into the feeder unit. Finally, constraint (5) requires that boxes which have components assigned to them are marked as allocated in the group.

Some additional constraints could be added to limit the search space, like:

$$\sum_{j=1,\dots,|C|} x_{jk} \leq \sum_{j=1,\dots,|C|} x_{j,k+1} \quad \text{for all } k = 1, \dots, K - 1 \tag{7}$$

$$\sum_{b=1,\dots,|B|} z_{ibk} \leq 1 \quad \text{for all } i = 1, \dots, |E| \text{ and } k = 1, \dots, K \tag{8}$$

Constraint (7) says that the groups must be in order of increasing size. Constraint (8) demands that a component is placed in one box only in a group.

2.7 *A JGP-B example solution*

The following description is a simple example of a JGP-B instance and its solution (see figure 1):

- $E = \{e_1, \ldots, e_{10}\}$, where
  - $w(e_1) = w(e_2) = w(e_3) = w(e_4) = 1$;
  - $w(e_5) = w(e_6) = w(e_7) = 2$;
  - $w(e_8) = w(e_9) = 3$ and
  - $w(e_{10}) = 4$.
- $C = \{c_1, \ldots, c_5\}$, where
  - $c_1 = \{e_1, e_3, e_4, e_6, e_8, e_9\}$;
  - $c_2 = \{e_2, e_4, e_5, e_6\}$;
  - $c_3 = \{e_1, e_3, e_7, e_8, e_9\}$;
  - $c_4 = \{e_2, e_4, e_6, e_{10}\}$ and
  - $c_5 = \{e_1, e_3, e_6, e_8\}$.
- $B = \{b_1, \ldots, b_4\}$, where
  - $a(b_1) = \{1\}, a(b_2) = \{2\}, a(b_3) = \{3\}, a(b_4) = \{2, 4\}$;
  - $s(b_1, 1) = 1, s(b_2, 2) = 2, s(b_3, 3) = 3, s(b_4, 2) = 3, s(b_4, 4) = 5$;
  - $v(b_1) = 3, v(b_2) = 4, v(b_3) = 6, v(b_4) = 8$ and
  - $w(b_1) = 4, w(b_2) = 5, w(b_3) = 7, w(b_4) = 9$.
- $w_{\max} = 18$.
- $n_{\max} = 5$.

This problem has a minimal grouping $G = \{(g_1, B_1, \alpha_1), (g_2, B_2, \alpha_2)\}$, where

- $g_1 = \{c_1, c_3, c_5\}$;
- $B_1 = \{b_1, b_2, b_3\}$;
- for $E(g_1) = \{e_1, e_3, e_4, e_6, e_7, e_8, e_9\}$ we have
  - $\alpha_1(e_1) = \alpha_1(e_3) = \alpha_1(e_4) = b_1$;
  - $\alpha_1(e_6) = \alpha_1(e_7) = b_2$;
  - $\alpha_1(e_8) = \alpha_1(e_9) = b_3$;
- $g_2 = \{c_2, c_4\}$;
- $B_2 = \{b_1, b_2, b_4\}$;
- for $E(g_2) = \{e_2, e_4, e_5, e_6, e_{10}\}$ we have
  - $\alpha_2(e_2) = \alpha_2(e_4) = b_1$;
  - $\alpha_2(e_5) = \alpha_2(e_6) = b_2$ and
  - $\alpha_2(e_7) = \alpha_2(e_{10}) = b_4$.

To prove that the grouping G is a solution to the described JGP-B instance, the formulation of the problem requires that it is feasible and of minimal cardinality. The box multisets B_1 and B_2 are feasible with respect to $n_{\max}$ and $w_{\max}$, since

- $|B_1| = |B_2| = 3 \leq n_{\max} = 5$;
- $w(B_1) = 4 + 5 + 7 = 16 \leq w_{\max} = 18$ and
- $w(B_2) = 4 + 5 + 9 = 18 \leq w_{\max}$.

Group g_1 is feasible with respect to $B_1 = \{b_1, b_2, b_3\}$, since

- $\sum_{e \in \alpha_1^{-1}(b_1)} s(b_1, w(e)) = 3 * s(b_1, 1) = 3 \leq v(b_1) = 3$;
- $\sum_{e \in \alpha_1^{-1}(b_2)} s(b_2, w(e)) = 2 * s(b_2, 2) = 4 \leq v(b_2) = 4$ and
- $\sum_{e \in \alpha_1^{-1}(b_3)} s(b_3, w(e)) = 2 * s(b_3, 3) = 6 \leq v(b_3) = 6$.

Finally, group g_2 is feasible with respect to $B_2 = \{b_1, b_2, b_4\}$, since

- $\sum_{e \in \alpha_2^{-1}(b_1)} s(b_1, w(e)) = 2 * s(b_1, 1) = 2 \leq v(b_1)$;
- $\sum_{e \in \alpha_2^{-1}(b_2)} s(b_2, w(e)) = 2 * s(b_2, 2) = 4 \leq v(b_2)$ and
- $\sum_{e \in \alpha_2^{-1}(b_4)} s(b_4, w(e)) = s(b_4, 2) + s(b_4, 4) = 8 \leq v(b_4) = 8$.

The analysis above shows that grouping G is feasible because both g_1 and g_2 are feasible. It is also of minimal cardinality because placing the PCBs to a one group g_{all} would require a box set B_{all} for which $\{b_1, b_3, b_4\} \subseteq B_{\text{all}}$, otherwise the boxes of B_{all} are not able to hold components of all widths given in the problem instance definition. This kind of box set does not fit in the feeder (due to $w_{\max}$) which means that no grouping of cardinality 1 is feasible.

3. Heuristic solution algorithms

This section considers four heuristic algorithms for JGP-B. The first one is a simple greedy algorithm which serves as a basis for the comparison. The other three algorithms use operations of merging, redistribution and splitting. These have been successful in previous clustering algorithms and this study therefore adapts them to solve the hard combinatorial optimization problem in question.

The difficulty in the JGP-B is that it includes two hard decisions to be made simultaneously. One is the selection of a job group spanning a certain set of components and the other is the selection of suitable boxes for the components of each component set. One way to partially overcome this complexity is to consider these two problems in separation. Therefore, the idea in the three algorithms is to alternate between the search for advantageous box multisets and groupings of jobs. The aim is that both groupings and box multisets adapt to the restrictions placed on the other. This idea is visible especially in algorithms MergeG and SplitG. In addition to this, the third algorithm RedistrG uses redistribution operations and applies some randomization while enhancing a grouping by moving and swapping jobs between groups.

In order to aid the understanding of the four grouping algorithms this section first introduces two methods for the selection and improving of box sets. The first one is a brute-force method which simply checks all feasible (maximal) subsets of B. The second is an improvement method based on local search. The algorithm using merge operations for grouping (MergeG) is parameterized to use either of these routines.

3.1 *Searching and improving box multisets*

The grouping algorithms of the next sections utilize an improvement algorithm which tries to recover from a situation where a grouping attempt runs out of feeder capacity, *i.e.* at least one group $g_i \in G$ is not feasible[†] with respect to B_i. We propose here a brute force and a local search algorithm for this purpose.

3.1.1 Brute force search for a feasible box multiset (BFBoxes). This approach is applicable when the size of the box multiset B is small. The simplest way would be to consider all (feasible) subsets of B, but BFBoxes proceeds slightly more intelligently because it considers *maximal* subsets only. A subset B' of B is *maximal*, if $w(B') \leq w_{\max}$, $|B'| \leq n_{\max}$ and

[†]We return to the test of feasibility in a while, but at this point it is supposed that one can simply decide whether a group is feasible or not.

either $|B'| = n_{\max}$ or $w(B') + w(b) > w_{\max}$ for all $b \in B \backslash B'$. It is easy to verify that if there exists some $B_i \subseteq B$ such that group g_i is feasible with respect to B_i, it is always possible to extend B_i to a maximal subset B' of B by simply adding new boxes into it as long as possible. It thus suffices to consider maximal subsets, only. However, even this modification may leave an unbearable amount of candidates left; for example in the case where each $b \in B$ has $v(b) = w(b) = 1$ and each $e \in E$ has $w(e) = 1$, we have $\binom{|B|}{w_{\max}}$ maximal subsets to consider. Function BFBoxes considers all maximal subsets B' of B and it returns the first B' which is found to be feasible with g_i. If there is no such B', the function returns the empty set.

3.1.2 Heuristic improvement of the box multiset (HIBoxes).

Let us denote by $U(g_i, B_i)$ the minimum number of unplaced components of group g_i when using the feeder box multiset B_i. This number is hard to determine exactly in general due to the inherent complexity of the bin-packing problem which should be solved for flexible boxes. However, the number of unplaced components is used only in a heuristic way when evaluating the different box multisets. The HIBoxes therefore is content to search for a rough approximation of the minimal number of components left over when filling B_i in an optimal way. It approximates this value by considering all boxes $b \in B_i$ in order of ascending $|a(b)|$ (*i.e.* fixed boxes are examined first) and by placing the components of $E(g)$ into each b in turn (placement of e into b is possible if $w(e) \in a(b)$ and there is enough free capacity left in b).

The HIBoxes uses these U-values to facilitate the process of transforming infeasible groups or box multisets to feasible; a group or a box multiset with the minimal violation of feasibility (*i.e.* the smallest U-value) is chosen. The same action gives also a heuristic answer whether group g_i is feasible with respect B_i. We have implemented this check as routine IsFeasible which returns 'true' if $U(g_i, B_i) = 0$ and 'false' otherwise.

The heuristic improvement algorithm HIBoxes tries to improve a given box multiset B_i by swapping boxes between B_i and $B \backslash B_i$. Note that the selection of single boxes is a greedy action and HIBoxes might get better results by changing several boxes at a time. Routine HIBoxes considers all possible swappings of boxes $b_1 \in B_i$ and $b_2 \in (B \backslash B_i)$ and computes the improvement

$$U(g_i, B_i) - U(g_i, B_i \backslash \{b_1\} \cup \{b_2\})$$

for each such operation. A swap operation giving the maximal improvement is chosen, and the swap is made if it releases capacity in the feeder. Swapping of boxes may also release empty space in the feeder. In that case HIBoxes tries to fill the free space with one or more boxes. The selection of a new box to box multiset B_i uses the following simple rule: select $b \in B \backslash B_i$ such that $B_i \cup b$ is feasible and $U(g_i, B_i) - U(g_i, B_i \cup b)$ is maximal. This rule (implemented as routine SelectBox) attempts to maximize the local improvement in the remaining unplaced components in a greedy way. The whole improvement process is iterated until no changes can be made.

3.2 *Greedy grouping (GreedyG)*

The greedy grouping starts by forming a group of a single job and augmenting it iteratively with new jobs according to the order of component similarities between remaining jobs and the group constructed so far. A new group is created when the augmentation of the present group would violate the feasibility constraint. The feasibility of a candidate group is checked by the BFBoxes algorithm of section 3.1.

The GreedyG algorithm gets as an input a set of jobs (C), a set of boxes (B), feeder unit width ($w_{\max}$) and a maximal allowed number of boxes in the feeder unit ($n_{\max}$). The result is a feasible job grouping (G). The method is as follows:

- Pick a random $c \in C$ and create a new singular group $g = \{c\}$.
- Sort all unplaced jobs $c' \in C$ according to $|c' \cap E(g)|$ into a descending order.
- Consider each c' in this order and add c' into g if there is a feasible box multiset for the augmented group $\{c'\} \cup g$.
- Iterate the same process for the remaining jobs.

Note that although this greedy algorithm may be too tedious to be performed manually, its basic idea should be reasonable to any human operator.

3.3 *Grouping by merging (MergeG)*

The method of this section uses the well-known technique of hierarchial clustering, which has turned out to be effective in practice (Kaukoranta 2000). What makes the situation with JGP-B radically different and more difficult are the two feeder unit constraints stating the maximal number of boxes and the possible box multisets. The MergeG therefore applies the ideas from clustering as a starting point only.

The grouping process begins by creating an initial grouping of $|C|$ singular groups for which a feasible multiset of boxes is determined with the brute force method (BFBoxes). It then merges these groups greedily as long as possible. The MergeG first calculates the intersections of the component sets for each group pair to select a proper order of merging operations, and considers the feasibility of the merge operations in the descending order of the cardinalities of these intersection sets. The search for a feasible box multiset is done at the merge steps using either the brute force method or the heuristic improvement method of section 3.1. This choice is given by parameter r ($r =$ BFBoxes or HIBoxes). The first feasible merge is performed and the same process (including the re-computation of the intersections) is iterated until no feasible merge is possible.

The algorithm MergeG gets as its input parameters r, C, B, $w_{\max}$ and $n_{\max}$. The result of the algorithm is either a feasible final grouping G or the empty set if no grouping was found. The implementation is outlined in figure 2.

For the sake of simplicity, the implementation of MergeG does not pay much attention to the optimization of the running time. The intersections, for example, are determined on each iteration and the result of a merge operation is recalculated even if no changes were made to the group pair since the previous iteration of the algorithm.

3.4 *Grouping by redistributing (RedistrG)*

While the MergeG-technique joins whole groups and preserves the feasibility at each step, it is possible to fill the free space in the boxes more carefully. RedistrG does this by moving individual jobs instead of groups, and by repairing groups which lose their feasibility (because of the redistribution) back to feasible. The idea is related to the JGP-algorithm ‘HG3’ of Knuutila *et al.* (2001).

Function RedistrG gets as its input a feasible initial grouping G (*e.g.* one created with MergeG) and a bound t for its execution time. The aim of RedistrG is to decrease the number of groups of G. It selects a random group from G and redistributes its jobs randomly to the remaining groups. If the resulting grouping is feasible, it repeats the same process; otherwise it tries to repair the infeasible grouping by alternating between changes of the grouping and the

```
function MERGEG($C, B, w_{max}, n_{max}, r$): Grouping
    $G = \{\}$
    for $i = 1..|C|$ do
        $g_i = \{c_i\}$
        $B_i$ = BFBOXES($g_i, B, w_{max}, n_{max}$)
        if $B_i = \{\}$ then return $\{\}$
        $G = G \cup \{(g_i, B_i)\}$
    repeat
        improved = False
        % compute and store the sizes of all pairwise group intersections
        Pairs = $\emptyset$
        for $i = 1..|G| - 1$ do
            for $j = i + 1..|G|$ do
                Pairs = Pairs $\cup\{(|E(g_i) \cap E(g_j)|, i, j)\}$
        % sort Pairs into descending order according to the intersection sizes
        SORT(Pairs)
        for $(n, i, j) \in$ Pairs do
            $g' := g_i \cup g_j$
            % try to find a feasible box multiset for the merged group $g'$
            if $r$ = BFBOXES then
                $B'$ = BFBOXES($g', B, w_{max}, n_{max}$)
            else
                % try to improve $B_i$ and, if that fails, $B_j$
                $B'$ = HIBOXES($g', B_i, w_{max}, n_{max}$)
                if not ISLEGAL($g', B'$) then
                    $B'$ = HIBOXES($g', B_j, w_{max}, n_{max}$)
            if ISLEGAL($g', B'$) then
                $G = G \backslash \{(g_i, B_i), (g_j, B_j)\} \cup \{(g', B')\}$
                improved = True
                break
    until not improved
    return $G$
```

Figure 2. Pseudocode for 'grouping by merging' algorithm MERGEG.

box multisets (the latter task is done with routine HIBOXES). If this process leads to a feasible grouping, REDISTRG re-iterates the above steps. If, however, the grouping remains infeasible, it restarts from the previous feasible grouping and selects another group to be distributed. The algorithm terminates after a given amount of time (t) and it returns the smallest feasible grouping found.

Figure 3 contains a pseudocode of REDISTRG. Here, UNPLACEDCOMPONENTS sums up the U-function values of all group-box multiset pairs in G. Routine MOVEJOBS moves jobs from an infeasible group to other groups until no moves preserving the feasibility of the target groups are possible. MOVEJOBS tries the move-outs greedily by using the original lexical ordering of the jobs and groups as the trial order.

3.5 *Grouping by splitting (SPLITG)*

The MERGEG (and consequently REDISTRG) performs agglomerative clustering by moving from a large number of groups to a smaller one. The SPLITG turns the idea of MERGEG around by starting with a large group consisting of all jobs. This is then split into smaller parts until a feasible grouping is found. The SPLITG has two decisions to consider: the selection of a group to split and performing the actual splitting.

A natural heuristic to select a group to split is once again based on the U-function. Supposing that grouping G is infeasible, SPLITG first selects the 'most infeasible' group of G, *i.e.* the one with a maximal U-value. The actual splitting of the group is performed so that the two new

```
function REDISTRG($G, C, B, w_{max}, n_{max}, t$): Grouping
    while the time bound $t$ is not exceeded do
        % select a 'source' group $g_s$ in random
        $(g_s, B_s) = $ RANDOMMEMBER($G$)
        $G' = G \backslash \{(g_s, B_s)\}$
        % redistribute the contents of $g_s$ into randomly selected groups
        for $c \in g_s$ do
            $(g_t, B_t) = $ RANDOMMEMBER($G'$)
            $g_t = g_t \cup \{c\}$
        %$G'$ is quite likely not feasible, try to improve it
        repeat
            $v_1 = $ UNPLACEDCOMPONENTS($G'$)
            for $(g_i, B_i) \in G'$ do
                $B_i = $ HIBOXES($g_i, B_i, B, w_{max}, n_{max}$)
            MOVEJOBS($G'$)
            $v_2 = $ UNPLACEDCOMPONENTS($G'$)
        until $v_1 \le v_2$
        if ISFEASIBLE($G'$) then $G = G'$
    return $G$
```

Figure 3. Pseudocode for 'grouping by redistributing' algorithm REDISTRG.

groups are approximately of the same size, and their contents are assigned in randomly. New groups start with the same box multiset as the split group had originally.

A pseudocode of SPLITG is shown in figure 4. Routine SELECTGROUPTOSPLIT selects the 'most infeasible' group, routine SPLITGROUP implements the splitting process and routine SWAPJOBS considers all swappings of job pairs such that one member of the swap is in an infeasible group and the other member is in a feasible group. The group pair which gives the maximal improvement based on the U-value is chosen for splitting.

4. Experimental results

We generated a set of test data with the same charasteristics as real data. Generated data was used in order to perform statistical tests on the performance of different solution algorithms; a large number of observations is needed to draw any conclusions of statistical significance. The test runs were performed on Pentium 4 2.0 GHz with 512 Mb main memory.

```
function SPLITG($C, B, w_{max}, n_{max}$): Grouping
    $G = \{(\{C\}, \emptyset, \emptyset)\}$
    while not ISFEASIBLE($G$) do
        MOVEJOBS($G$)
        SWAPJOBS($G$)
        for $(g_i, B_i) \in G$ do
            $B_i = $ HIBOXES($g_i, B_i, B, w_{max}, n_{max}$)
        if not ISFEASIBLE($G$) then
            MOVEJOBS($G$)
            SWAPJOBS($G$)
        if not ISFEASIBLE($G$) then
            $(g_i, B_i) = $ SELECTGROUPTOSPLIT($G$)
            $g', g'' = $ SPLITGROUP($g_i$)
            $G = G \backslash \{(g_i, B_i)\} \cup \{(g', B_i), (g'', B_i)\}$
    return $G$
```

Figure 4. Pseudocode for 'grouping by splitting' algorithm SPLITG.

4.1 *Test problems*

A component library containing the data of *ca.* 4000 electronic components was used when generating the test data. The set of components E was created by selecting randomly 18% (583) of these (to keep running times acceptable). After this we created a master set of 300 test jobs by first choosing for each c_i $(i = 1, \dots, 300)$ the size $|c_i|$ randomly from a user-defined interval (*e.g.* $[12, \dots, 30]$) and then assigning the drawn amount of different randomly chosen components to c_i. The problem instances were finally formed by sampling (without repetitions) the desired number of different jobs from this generated set. When comparing the heuristics to each other, we used sample sizes of $|C| = 30$ and $|C| = 60$, and the results were averaged over 40 different samples. A comparison to optimal solutions was possible for smaller problem instances ($|C| = 8$ with 24 components in each; 30 problem instances).

The generation of boxes was based on the occurrence frequencies of different component widths in PCBs. The following procedure was used to generate box multisets:

1. Compute the frequencies $f(w, c)$ of different component widths for each $w \in w(E)$ and $c \in C$: $f(w, c) = |\{e \in c \mid w(e) = w\}|$.
2. Compute for each width $w \in w(E)$ its maximal frequency $f_{\max}(w)$ within C: $f_{\max}(w) = \max\{f(w, c) \mid c \in C\}$.
3. Sort these maximal frequencies into descending order $f_1, f_2, \dots$.
4. Let $f_1 = f_{\max}(w)$. Create 3 boxes of fixed type with capacity $w * f_{\max}(w)$.
5. Let $f_2 = f_{\max}(w_i)$ and $f_3 = f_{\max}(w_j)$. Create 2 boxes of fixed type with capacity $w_i * f_{\max}(w_i)$ and 2 boxes of capacity $w_j * f_{\max}(w_j)$.
6. Create one box of fixed type for all remaining widths w with capacity $2 * w * f_{\max}(w)$.

The flexible boxes were created such that one flexible box can hold components from a certain width range and these ranges do not overlap.

We let $w(E) = \{1, 2, 3, 4, 5, 7, 9\}$, and the multiset of boxes derived using the method just described be

$$B = \{b_1, b_1, b_1, b_2, b_2, b_3, b_3, b_4, b_5, b_6, b_7, b_8, b_8, b_9, b_{10}\}$$

where boxes types b_1 to b_7 are fixed and b_8, b_9 and b_{10} are flexible. The properties $a(b_i)$, $v(b_i)$ and $w(b_i)$ for $i = 1, \dots, 10$ are given in table 1 for each box type b_i. Note that the numbers $v(b_i)$ and $w(b_i)$ use different scales (otherwise the internal capacity would exceed the external one). Table 1 contains also the space allocations $s(b_i, w)$ of components. The total width $w_{\max}$ of the feeder unit was 9 and the maximal number of boxes $n_{\max}$ was 9, too.

Table 1. Box types used in the tests.

Box	a	v	w	s
b_1	1	24	1	1
b_2	2	16	1	2
b_3	3	36	1	3
b_4	4	28	1	4
b_5	5	30	1	5
b_6	7	14	1	7
b_7	9	18	1	9
b_8	$\{1, 2, 3\}$	12	1	$\{1.3, 2.3, 3.4\}$
b_9	$\{4, 5\}$	15	1	$\{4.5, 5.3\}$
b_{10}	$\{7, 9\}$	15	1	$\{8.1, 10.5\}$

Note: The space allocations s for flexible boxes are given in the same order as the respective widths in column a.

Table 2. A sample of grouping sizes for $|C| = 30$, test 1.

Test	MERGEG(H)	REDISTRG	SPLITG	GREEDYG	MERGEG(B)
1	5	5	5	6	5
2	6	5	5	7	5
3	6	6	6	7	6
4	5	5	5	7	5
5	6	5	6	7	6
⋮	⋮	⋮	⋮	⋮	⋮
40	6	5	6	7	6
Average	5.28	5.13	5.4	6.4	5.25
Standard deviation	0.45	0.40	0.50	0.59	0.44

Table 3. A sample of grouping sizes for $|C| = 60$, test 2.

Test	MERGEG(H)	REDISTRG	SPLITG	GREEDYG
1	9	9	9	13
2	9	9	10	12
3	9	9	9	13
4	9	9	9	11
5	9	9	9	12
⋮	⋮	⋮	⋮	⋮
40	9	9	10	12
Average	9.2	9.05	9.65	12.18
Standard deviation	0.61	0.50	0.62	0.75

4.2 *Experiments*

In what follows, the notations MERGEG(H) and MERGEG(B) denote the use of MERGEG with the options $r =$ HIBOXES and $r =$ BFBOXES, respectively. For small problem instances ($|C| = 8$) we evaluated the heuristics against optimal solutions. Because optimal solutions were not available for large problem instances ($|C| = 30$ and $|C| = 60$), we compared the results of GREEDYG, MERGEG(H), MERGEG(B), REDISTRG[†] and SPLITG against each other. We performed the following six sets of tests:

1. Each problem instance consisted of 30 jobs and the grouping was repeated 40 times for different problem instances, see table 2 for the results.
2. Each problem instance consisted of 60 jobs, and the grouping was again performed to see the effect of the problem size on the number of groups and the running times for different solution algorithms, see table 3. The results of MERGEG(B) have been omitted due to the large running times of the method.
3. This test was created to measure the effect of the number of different boxes on the running time of MERGEG (using both brute force and heuristic box set improvement technique). The test problems were the same as in the test 1, but the box multiset was increased to contain 2 new flexible boxes of type b_8 and one of type b_9, see table 4.
4. The purpose of this test run was to measure the running time of MERGEG as a function of $|B|$ for BFBOXES and HIBOXES, see table 5.
5. In this test run we observed the grouping sizes as a function of the timelimit set for the REDISTRG method, see table 6.

[†]Recall that the initial grouping of REDISTRG is computed by MERGEG(H).

Table 4. A sample of grouping sizes and running times for large box multisets, test 3.

	MergeG(H)		MergeG(B)	
Test	$\|G\|$	Time (s)	$\|G\|$	Time (s)
1	5	2	5	261
2	6	3	5	315
3	6	4	6	297
4	5	4	5	272
5	6	2	6	265
⋮	⋮	⋮	⋮	⋮
40	6	4	6	311
Average	5.28	3.15	5.23	287

Table 5. Average running times of the algorithms, test 4.

	Average running time	
Algorithm	Test 1	Test 2
MergeG(H)	2.1	7.2
RedistrG	32.1	67.1
SplitG	2.2	27.4
GreedyG	1.3	2.5

6. The initial grouping given to the RedistrG method affects the final result. We measured the grouping sizes when using both the MergeG and SplitG methods to create the initial grouping. See table 7.
7. The MILP formulation of JGP-B was implemented with ILOG CPLEX. The heuristic results given by RedistrG were compared against the optimal ones for small problem instances. Table 8 shows the results.

The results of tables 2 and 3 show that RedistrG gives the best groupings for the test problems of this study. The differences from the other two algorithms are relatively small but still systematic.

When testing the statistical significance of these results (with a paired t-test) we observed that for problems with 30 jobs RedistrG gave statistically significantly ($p = 2.04\%$) smaller groups than MergeG(H) and SplitG ($t = 2.62$ and $t = 3.44$, respectively). RedistrG was able

Table 6. The grouping sizes given by RedistrG as a function of time.

	Algorithm				
Test	MergeG	RedistrG(30)	RedistrG(60)	RedistrG(120)	RedistrG(240)
1	4	4	4	4	4
2	5	4	4	4	4
⋮	⋮	⋮	⋮	⋮	⋮
40	5	5	5	5	5
Average	4.60	4.50	4.40	4.38	4.38
Standard deviation	0.50	0.51	0.50	0.49	0.49

Note: Time limits are given in seconds inside the parenthesis. The number of jobs is 30.

Table 7. Comparing the results of REDISTRG when using MERGEG and SPLITG as the initial grouping.

	Algorithm			
Test	MERGEG	SPLITG	REDISTRG(MERGEG)	REDISTRG(SPLITG)
1	4	5	4	4
2	5	5	5	4
⋮	⋮	⋮	⋮	⋮
40	5	5	5	5
Average	4.60	4.65	4.48	4.38
Standard deviation	0.50	0.48	0.51	0.49

Note: Time limit of REDISTRG was 4 minutes, and the number of jobs was 30.

to improve the initial grouping (given by MERGEG(H)) in 5 cases out of 40. The same observation holds also for the larger (60 jobs) problem instances, moreover algorithm MERGEG(H) was now significantly better than SPLITG ($t = 4.77$). The greedy algorithm was clearly worse than the other three algorithms, as expected. There were, however, 4 cases out of 40 where the results of MERGEG(B) and the greedy algorithm were the same. We note that the results of MERGEG(B) and MERGEG(H) do not differ statistically for the smaller problem size, although there are cases where the two algorithms find groupings of different sizes (in both directions).

The results of table 4 clearly show that the brute force search is much slower than the heuristic improvement technique even with relatively small box multisets. The increase in the running time is (as expected) exponential when the size of the box multiset increases. Although the exhaustive search gives smaller groupings for some test cases, the difference is not significant ($t = 1.00$).

Table 5 shows that the good results of REDISTRG are paid for in larger running times. Recall that one can adjust the execution time of REDISTRG, so the running times may be shorter in practice. In comparison to MERGEG(H), the running times of REDISTRG are almost tenfold, while the solutions are close to each other for these two methods. Note, however, that even a small improvement (in grouping size) is often a remarkable achievement in the actual production time. Finally, method SPLITG seems to be quite sensitive to the problem size.

One can observe from figure 5 the exponential increase in the execution time of BFBOXES with respect to the box set size $|B|$. MERGEG with HIBOXES shows near linear running times with respect to box set size as expected.

It is observed from table 6 that the average grouping size decreases until the time limit reaches two minutes. For practical situations this is not a limiting factor for the use of REDISTRG

Table 8. Comparing the results of the MERGEG and REDISTRG against the optimal solutions calculated with ILOG CPLEX.

	Algorithm		
Test	MERGEG	REDISTRG	Optimal
1	3	3	3
2	4	4	4
⋮	⋮	⋮	⋮
30	3	3	3
Average	3.3	3.2	3.0
Standard deviation	0.60	0.55	0.50

Note: Problem instances consist of 8 jobs, 24 components and 4 feeder boxes.

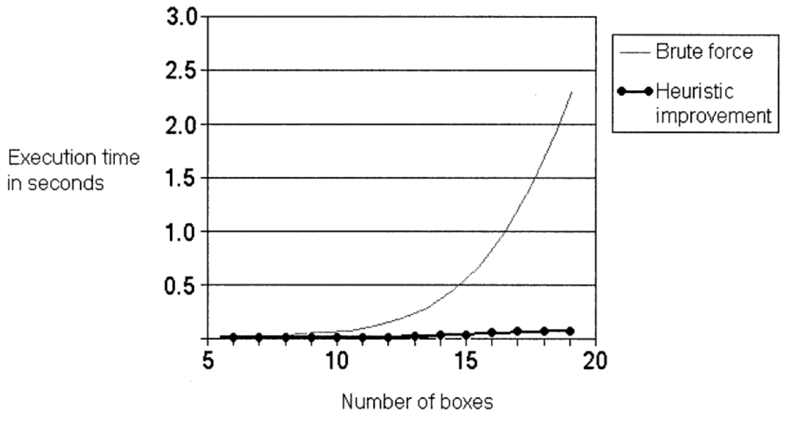


Figure 5. Running time comparison of MERGEG when BFBOXES or HIBOXES.

method. The running times could easily be scaled with a factor of 10 and the methods would still be useful.

It is observed from table 7 that there is no statistical difference when using MERGEG or SPLITG in REDISTRG. The slight difference in the results might originate from the fact that the groups created by SPLITG are already 'redistributed' in the sense that it moves singular jobs in order to make the grouping legal. Conversely, MERGEG combines larger blocks of jobs and this tends to leave more empty space in the feeder configuration.

We were able to solve optimally one problem instance of size 10 jobs and 30 components within the time limit of three hours. We also considered 30 problem instances with 8 jobs and 24 components. Out of these problems we managed to solve 29 within a time limit of one hour. In this test run 87% of the solutions found by REDISTRG were also optimal, see table 8. When considering the fast increase in the number of decision variables with respect to the number of jobs and components, it is highly unlikely that problems of practical size (*e.g.* 30 jobs, 200 components) could be solved exactly within reasonable timelimits.

5. Concluding remarks

The paper defined a variant of the job grouping problem in which the feeder unit of a component placement machine is organized as a bed for a set of feeder boxes. The boxes are removable, and each of them has a specific capacity and width. Because of the relatively high cost of these boxes, there is usually only a certain limited storage of different boxes available. Boxes come in two main types: fixed and flexible. The aim in JGP-B is to form a minimal size grouping from a set of PCB jobs such that the components of the PCB types in a group fit in the boxes and that the boxes fit in the feeder unit at the same time. The problem is clearly a generalization of the common job grouping problem in PCB assembly and it appears in the production control of modern component placement machines. The main new issue here is the selection of a suitable box multiset that increases the complexity of the problem significantly.

Three heuristic algorithms were given for the JGP-B by utilizing the general ideas from clustering and the common JGP-algorithms. The improvement of the selected box multiset was a cornerstone in these algorithms. Both the brute-force and heuristic algorithm were proposed for this purpose. The results of the experimental tests show that in the case of 60 jobs the naive algorithm GREEDYG gives in an average 35% weaker results than the best proposed method

(RedistrG). In addition, the brute force method for box selection turned out to be very slow due to the large number of possible box configurations.

The differences between the various techniques were seemingly rather small when the number of groups is concerned. One should, however, keep in mind that the cost of the extra work caused by even a single group is high.

There are several areas for further development in this research. In particular, implementation decisions should be analyzed in more detail and we could consider other alternatives for the given algorithms. Below are a few examples:

- The heuristic algorithm used to verify the feasibility of a grouping (function IsFeasible) is quite simple. A more involved technique might use the feeder capacity more tightly and give in some cases smaller groupings.
- Our earlier JGP implementation of many repair routines (like function SwapJobs, see Knuutila *et al.* (2001)) allowed operations even if some of the groups did not remain feasible. A similar approach could be applied to the JGP-B problem, too.
- Function SplitG divides groups so that the PCB jobs are placed randomly into the target groups. Another possibility would be to place similar jobs into the same new group.

One unaddressed, but yet important, research problem is to combine JGP-B with finding the smallest (or least expensive) box multiset from some global set. This would give a valuable aid to production engineers when deciding which boxes to acquire for the production. As mentioned in the introduction, the production control situation contains several additional aspects which have been omitted in the research of JGP, JGP-T and JGP-B. Their consideration could be handled by hybridizing the JGP formulation with the traditional tool switching problem. This would lead to a computationally hard but interesting combinatorial optimization problem.

It is hard to make precise comparisons between the different settings of JGP. We have previously solved problems of similar size (30 jobs) with linear, uniform feeder units and found typically minimal groupings of sizes 3 and 4, while their number was between 4 and 5 for JGP-T. With JGP-B the groupings found are typically of size 5 or 6. A natural reason for this effect is the increase in the demands caused by the various restrictions added. One should thus avoid the too simple conclusion that the increase is due to weaker algorithm design or that this kind of equipment should be avoided in general: the more sophisticated machines increase significantly the usability of the production facilities in other respects.

References

Bhaskar, G. and Narendran, T.T., Grouping PCBs for set-up reduction: A maximum spanning tree approach. *Int. J. Prod. Res.*, 1992, **34**(3), 621–632.

Crama, Y., Combinatorial optimization models for production scheduling in automated manufacturing. *Eur. J. Oper. Res.*, 1997, **99**(1), 136–153.

Crama, Y., Oerlemans, A. and Spieksma, F., *Production Planning in Automated Manufacturing*. *Lecture Notes in Economics and Mathematical Systems*, 1994, Volume 414 (Springer-Verlag).

Crama, Y. and van de, Klundert, J., Worst-case performance of approximation algorithms for tool management problems. *Nav. Res. Log.*, 1999, **46**(5), 445–462.

Crama, Y., van de Klundert, J. and Frits Spieksma, C.R., Production planning problems in printed circuit board assembly. *Discrete App. Math.*, 2002, **123**, 339–361.

Hirvikorpi, M., Knuutila, T., Johnsson, M. and Nevalainen, O. (2003) Grouping of PCB assembly jobs in the case of flexible feeder units. *Technical Report 505*, Turku Centre for Computer Science.

Johnsson, M., Operational and tactical level optimization in printed circuit board assembly. *PhD thesis*, University of Turku, 1999 (TUCS Dissertation 16).

Johtela, T., Smed, J., Johnsson, M. and Nevalainen, O., Fuzzy approach for modeling multiple criteria in the job grouping problem. *In Proceedings of the 25th International Conference on Computers & Industrial Engineering*, edited by M.I. Dessoyky, S.M. Waly and M.S. Eid (New Orleans, LA, 1988), pp. 447–450.

Kaukoranta, T., Iterative and hierarchical methods for codebook generation in vector quantization. PhD thesis, University of Turku, 2000 (TUCS Dissertation 22).

Knuutila, T., Puranen, M., Johnsson, M. and Nevalainen, O., Three perspectives for solving the job grouping problem. *Int. J. Prod. Res.*, 2001, **39**(18), 4261–4280.

Knuutila, T., Hirvikorpi, M., Johnsson, M. and Nevalainen, O., Grouping PCB assembly jobs with typed component feeder units. *Int. J. Flex. Manuf.*, 2004, **16** (submitted for publication).

Landers, T.L., Brown, W.D., Fant, E.W., Malstrom, E.M. and Schmitt, N.M., *Electronics Manufacturing Processes*, 1994 (Prentice-Hall: Englewood Cliffs, NJ).

Leon, V.J. and Peters, B.A., Replanning and analysis of partial setup strategies in printed circuit board assembly systems. *Int. J. Flex. Manuf. Syst.*, 1996, **8**(4), 389–412.

Martello, S. and Toth, P., *Knapsack Problems* (John Wiley & Sons).

McGinnis, L.F., Ammons, J.C., Carlyle, M., Cranmer, L., DePuy, G.W., Ellis, K.P., Tovey, C.A. and Xu, H., Automated process planning for printed circuit card assembly. *IIE Trans.*, 1992, **24**(4), 18–30.

Shtub, A. and Maimon, O., Role of similarity in PCB grouping procedures. *Int. J. Prod. Res.*, 1992, **30**(5), 973–983.

Smed, J., Production planning in printed circuit board assembly. PhD thesis, University of Turku, 2002 (TUCS Dissertation 36).

Smed, J., Johnsson, M., Puranen, M., Leipälä, T. and Nevalainen, O., Job grouping in surface mounted component printing. *Robot. Comput.-Integr. Manuf.*, 1999, **15**(1), 39–49.

Tang, C.S. and Denardo, E.V., Models arising from a flexible manufacturing machine. *Oper. Res.*, 1988, **36**(5), 767–784.

Publication III

Hirvikorpi M., Salonen K., Knuutila T. and Nevalainen O.S., The general two level storage management problem: a reconsideration of the KTNS-rule. To appear in *European journal of operational research*.

Available online at www.sciencedirect.com

SCIENCE DIRECT®

European Journal of Operational Research xxx (2004) xxx–xxx

EUROPEAN JOURNAL OF OPERATIONAL RESEARCH

www.elsevier.com/locate/dsw

Production, Manufacturing and Logistics

The general two-level storage management problem: A reconsideration of the KTNS-rule ☆

Mika Hirvikorpi *, Kari Salonen, Timo Knuutila, Olli S. Nevalainen

Department of Information Technology and Turku Centre for Computer Science (TUCS), University of Turku, Lemminkäisenkatu 14 A, Turku 20520, Finland

Received 30 May 2003; accepted 23 August 2004

Abstract

A two-level storage management problem arising in flexible manufacturing systems is studied. We consider the case of several different types of PCBs (printed circuit board) to be processed with a component assembly machine. The PCB processing order is fixed and all components to be assembled to a PCB should be in the feeder of the machine before starting the processing of the PCB in question. Our task is to perform the component switches between the primary storage (feeder) and the secondary storage (component reel shelves) so that the overall switching cost of the component reels is minimal. The component reel switches are necessary because of the limited capacity of the machine feeder.

The "Keep Tool Needed Soonest" policy is known to be optimal when component reel widths are equal. A more general problem in which different component reels have different widths is considered here. In contrast to previous studies by Matzliach and Tzur [The online tool switching problem with non-uniform tool size, International Journal of Production Research 36 (12) (1998) 3407–3420] we assume that reorganizations of the feeder are necessary in order to place a wider component reel into the feeder where the free space is fragmented into smaller pieces. In PCB assembly the reorganization costs are substantial. The objective is then to minimize the sum of the costs for component switches between the two storage levels and the reorganizations of the feeder.

© 2004 Published by Elsevier B.V.

Keywords: Flexible manufacturing systems; PCB assembly industry; Tool change; Heuristics; Exact solution

☆ This work has been partially financed by the Graduate School, Turku Centre for Computer Science (TUCS).

* Corresponding author. Tel.: +358 2 3338690; fax: +358 2 3338600.

E-mail address: mika.hirvikorpi@utu.fi (M. Hirvikorpi).

0377-2217/$ - see front matter © 2004 Published by Elsevier B.V.
doi:10.1016/j.ejor.2004.08.031

1. Introduction

Increasing the production efficiency in flexible manufacturing systems can be approached from several different perspectives [3]. One possibility is to minimize the total time used for the tool changes, which leads to the so called *tool switching problem* [4]. In this problem a set of parts of different types are processed by a single machine. The machine processes the parts with some tools which vary from part to part. The proper tools should be in the tool magazine of the machine prior to starting the processing of a certain part. Each time a new tool is inserted into the magazine some manual actions must be taken. Tool changes are necessary due to the limited capacity of the tool magazine. In order to decrease this work one has to determine a processing sequence of the parts and a tool change policy which minimizes the total change work.

We can consider the management of the tool magazine as a two-level storage management (SM) problem where the tool magazine serves as a primary storage of limited capacity and the auxiliary storage is large. Instead of tools we can then speak of items to be stored. The term pairs (primary storage, item) and (tool magazine, tool) are therefore used interchangeably in the following text. The first pair of concepts underlines the general use of the problem while the second pair draws our attention to the concrete situation in flexible manufacturing.

As a concrete example consider the case of a single printed circuit board (PCB) assembly machine and several different types of PCBs to be processed. Each of these PCBs requires the insertion of certain types of components by the means of an automatic high speed machine. The machine has a feeder which holds all the components the machine can print directly. The feeder has a limited capacity and it is usually a lot smaller than the number of different component types the PCBs require in whole. This is why the machine's operation must be interrupted once in a while, new components must be loaded to the feeder and some of the old ones must be removed. One then has a situation equivalent to the tool switching problem of flexible machines. The order of the PCB assembly jobs must be selected so that the number of component switches in and out of the feeder is minimal.

Each PCB assembly job consists actually of a batch of bare PCBs of the same type, i.e. the layout of the printed circuits is fixed as well as the sets of electronic components to be assembled on them. This problem consists of two parts—in the first subproblem one must select the order in which the PCB assembly jobs are processed. After this has been done, the feeder change operations must be managed. This paper concentrates on the latter subproblem where we have actually again a two level storage of components. The machine can only insert components which are in the *primary storage* (feeder). Other components are supposed to reside in the shelves of the secondary storage in the vicinity of the machine area. Our problem is to minimize the total cost of moving the components when the sequence of processing is known in advance. This means that we have already selected the order of the PCB assembly jobs and we know which components are needed and when.

One way to classify the PCB assembly problems is to do it according to the number of PCB types and production machines in use [8]. The component change problem of this paper belongs to the multiple PCB, one machine—category (M-1). This category considers the setup strategies including unique, minimum, group and partial setup [11]. Our problem is a part of the minimum setup problem. Crama et al. [5] classify the PCB assembly problems to eight subclasses (SP1–SP8) [5]. The classes SP1–SP4 are for multiple machines, while classes SP5–SP8 are concerned with one PCB assembly machine problems but only with one PCB type. Our problem can not therefore be directly placed into this hierarchy.

The "Keep Tool Needed Soonest" (KTNS) policy was developed by Tang and Denardo [16] for the two-level SM problem and it has been shown to be optimal (see [4,16]). According to this policy, components are removed only when the storage becomes full and components which are needed latest in the future are removed. However, the rule is optimal only when handling components of the same size. In this paper we generalize the situation by considering a problem where components have *different sizes* (widths). This type

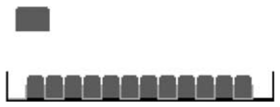


Fig. 1. Fragmented linear primary storage (feeder); insertion of a 2 slots wide item (component).

of problem was first considered by Matzliach and Tzur [13], who gave a proof for its NP-completeness and developed an optimal solution with integer programming along with heuristics. [1]

Our point of view still differs from that of [13] in the way of organizing the items in the primary storage. The abstraction made by Matzliach and Tzur assumed that the primary storage does not require any kind of costly reorganization during the process. They considered the primary storage only as a single number indicating its capacity. In the cases like the PCB assembly with fast chip shooter insertion machines we must also consider the placement of the components to the feeder which is arranged as an array of feeder slots. For example, Fig. 1 shows a situation where we want to insert a 2 slots wide component in a linearly ordered feeder and there are currently two feeder slots of size 1 free. In [13] it was no problem that the free slots are not neighbours in the storage. In the case considered here, one cannot put the new component to the feeder without moving or removing one of the components already in there. Therefore, the formulation of Section 2 takes into account the current placement of the components, too.

Our research problem is related to several other problems from manufacturing and systems programming. One way is to see it as a special kind of the well-known tool switching problem, see [6] for the case of uniform tool sizes (one magazine slot per tool) and *infinite tool lives*. We observe that the SM-problem studied in the present paper is *static* in the sense that all tool requests and their order are known prior to the management decisions. This assumption is valid in many practical cases but there are important situations where the problem is *dynamic* in nature. Matzliach and Tzur [12] consider the tool switching problem also in the dynamic situation, where the future requests are not known before they occur. This makes the tool management still much harder, and online algorithms for the problem have very weak worst case performance limits.

The two-level SM-problem is one form of *demand paging* appearing in the implementation of virtual memory systems. The task here is to select a victim page in the primary memory to make room for a new page that caused a page fault. A large set of paging algorithms have been proposed. They, however, are of the online type and the page frames are of uniform size. For more information, see standard textbooks on operating systems [15]. Management of *segmented computer memory* calls for a technique for allocating and deallocating data/program segments of variable sizes from a central memory of a limited size [9]. The problem is, however, of the dynamic type, only one segment is reserved in turn, and the reuse of segments is not considered. See also Mookerjee [14] for dynamic management of two-level storage in case of fixed sized segments.

Another practical situation of the tool management is the machine-level control of computer numerical control (CNC) machines, see the work by Avci and Akturk [2] for tool magazine management and sequencing of machine operations for a batch of jobs. One has, on this level of operation control, to decrease the tooling and tool operating costs while maintaining the feasibility of precedence, tool magazine capacity, tool lives and tool availability constraints.

Another related problem deals with the tool changes due to the wearing of tools (finite life times). When the order of the jobs is not fixed, one can schedule the jobs and plan the necessary tool changes so that the

[1] See [7] for a preliminary version of the present paper. Tzur and Altman [18] have published independently a work on a joint problem of making job sequencing, tool switching and tool locating decisions. For the two last subproblems they give heuristics which are very different from ours.

total average completion time of the jobs is minimized. Akturk et al. [1] suppose that the job processing times are constant and known a priori, but only one tool type with known constant life and unlimited availability is in use.

Tooling can also be seen in a wider context as a factor of the total costs of manufacturing. Turkcan et al. [17] consider a two objective scheduling model for multiple parallel CNC machines where the objectives are minimizing the manufacturing cost (including tooling costs) and the total weighted tardiness. The model proposes several different schedules depending on the weighting of the two objectives. Arrangement of the tool magazine is not considered explicitly in this important work.

The presentation is organized as follows. In Section 2 we give a formulation of the *general two-level storage management problem* (GSM-1) in the case of *single tool requests*, i.e. each PCB assembly job uses only one tool (component) at a time. Section 3 presents an integer programming formulation to the GSM-1-problem. In Section 4 we develop a new heuristic to solve the GSM-1-problem. Section 5 gives numerical results computed with the algorithms presented in [13] and with our methods. Section 6 introduces algorithms for the multitool request case (GSMM) and Section 7 summarizes the numerical results for our heuristics and the algorithms presented in [13]. The paper is closed with some conclusions in Section 8.

2. Problem definition of GSM-1

We assume that a set of parts $P(|P| = T)$ and set of tools I is given. Each part $j \in P$ should be processed by using a *single tool* $d_j \in I$ with a flexible production machine. The tools are changeable so that a subset of the tools used by all parts can be held in the tool magazine from which they can be picked up to the actual processing. The remaining subset of tools is stored in an auxiliary storage nearby the machine, see Fig. 2.

The tool magazine is organized to consist of K slots and each tool $i \in I$ (components in PCB assembly) reserves a certain number v_i of adjacent slots of the magazine, depending on the physical dimensions of the tool. In the context of PCB-assembly the tool magazine, which is called feeder, stores actually a set of components of each type. Each set is organized as a component reel, stick, etc., usually of large capacity. On the other hand, parts in PCB-assembly are batches of PCBs of identical layouts. This causes that component reels now and then run out of components (a "tool wears out"). However, the replenishment of the components is supposed to be a relatively fast operation and, unlike the CNC machines of metal industry, the

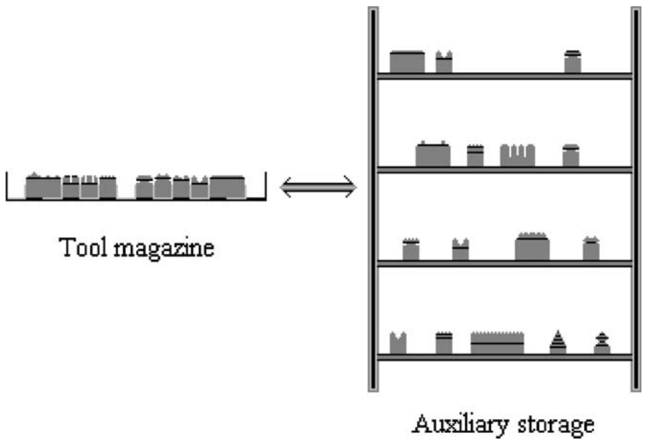


Fig. 2. The use of a linear tool magazine.

change of component reels or sticks can be made on demand while processing a PCB without any consequences to the quality of the product. In addition, the number of consumed components is independent on the feeder management politics. We can therefore omit the modeling and suppose infinite tool lives here for the component reels. In order to simplify the notations, if there is no risk of confusion, we next speak instead of component reels simply of components.

We further assume that the order of the parts has been fixed and all tools do not fit the magazine simultaneously. This latter assumption holds in typical manufacturing situations, where the usage profile of tools is commonly very skewed: some components are needed in majority of PCBs while there are several components placed on few PCB types only (note that the problem becomes trivial in the rare case where all tools fit the magazine simultaneously). With these assumptions the limited capacity of the magazine causes that we are forced to change tools in the magazine. This can only be done by removing some tools to make place for a new tool to be inserted. Because the tools are of different sizes it might be advantageous to remove two or more small tools to release slots for a larger tool. If these slots are non-adjacent we have a fragmented free space in the magazine which should be unified prior to the insertion of the new tool. We suppose that a transfer cost c_i is associated with each tool operation including removal, insertion and move of a tool. The *general two-level storage management problem with single tool request per job* (GSM-1) asks for a tool change policy which minimizes the total costs for tool management operations for a fixed ordering of the processing the parts.

Each tool i is expressed by two parameters: width v_i and transfer cost c_i. In order to add flexibility to the problem definition, we introduce three cost factors for different types of transferring operations: removal c_r, insertion c_a, and move c_m. These factors are then used for weighting the c_i-values at different update operations.

While the above parameters define an instance of the GSM-1-problem, a solution can be expressed as a sequence of the *primary storage* (i.e. magazine) *states* $S(t)$ ($t = 1, \ldots, T$) giving the subset of tools and their positions in the magazine when processing part t.

To sum up, an instance of the GSM-1-problem is defined by the following parameters:

$K \in \mathbb{N}$	magazine capacity as a number of slots;
$T \in \mathbb{N}$	number of tool requests;
$I \subset \mathbb{N}$	set of tools. All tools $i \in I$ have the following properties:
$c_i \in \mathbb{N}$	cost for transferring tool i;
$v_i \in \mathbb{N}$	capacity in number of magazine slots consumed by tool i;
$d_t \in I$	tool required at time $t = 1, \ldots, T$;
$c_a \in \mathbb{R}$	cost factor for insertion operation;
$c_r \in \mathbb{R}$	cost factor for removal operation;
$c_m \in \mathbb{R}$	cost factor for move operation.

Note that c_i may depend on size v_i, while c_a, c_r and c_m describe the relative costs of different storage operations.

A solution of the problem can be stated as a sequence $t = 1, \ldots, T$ of primary storage states. A primary storage state (PSS) $S(t)$ is an ordered pair (J_t, α_t), where:

- J_t is the set of tools in the primary storage at moment t ($J_t \subseteq T$).
- α_t gives the magazine (i.e. primary storage) assignment of the tools, i.e. it is an injective mapping $i \mapsto p$ where $p \in [1, K]$ for all $i \in J_t$ and it tells the leftmost magazine slot consumed by tool i. The magazine slots are indexed from 1 to K.

Our notation is similar to that of [13], except for the explicit definition of the primary storage states and the allowed changes between them. We observe that the notations can be easily translated to the PCB assembly terminology.

The inclusion of tool move costs ($c_m > 0$) causes an essential complication of the solution process because we must now ascertain that any two tools in the magazine do not collide, i.e. their slot allocations are disjoint. This is stated by the following property:

Primary storage state $S(t) = (J_t, \alpha_t)$ is *feasible* with respect to K and d_t, if and only if

(1) $\alpha_t(i) \in [1, K - v_i + 1]$ for all $i \in J_t$, meaning that all tools are in proper slots inside the magazine.
(2) Either $\alpha_t(i_1) + v_{i_1} \leqslant \alpha_t(i_2)$ or $\alpha_t(i_2) + v_{i_2} \leqslant \alpha_t(i_1)$ for all $i_1, i_2 \in J_t$ and $i_1 \neq i_2$, meaning that different tools in PSS do not use the same slots.
(3) $\sum_{i \in J_t} v_i \leqslant K$, i.e. the sum of tool sizes in PSS does not exceed the given capacity.
(4) $d_t \in J_t$, i.e. the tool used by part t is in the tool magazine at moment t.

Conditions 3 and 4 have been added to aid the description of the solution process. The first of these is redundant because it follows from conditions 1 and 2. The next notation expresses for two tool magazine states S_1 and S_2 those tools which are present in both states but in different slots. The concept is useful when calculating the transfer costs for tool moves.

Tool difference set $D(S_1, S_2)$ for primary storage states $S_1 = (J_1, \alpha_1)$ and $S_2 = (J_2, \alpha_2)$ is defined as follows:

$$D(S_1, S_2) = \{i \mid i \in J_1 \cap J_2 \text{ and } \alpha_1(i) \neq \alpha_2(i)\}.$$

We can now calculate the *switch cost between primary storage states* $S_1 = (J_1, \alpha_1)$ and $S_2 = (J_2, \alpha_2)$ as the sum of costs of the removed, moved and inserted tools:

$$c(S_1, S_2) = c_r \sum_{i \in J_1 - J_2} c_i + c_m \sum_{k \in D(S_1, S_2)} c_k + c_a \sum_{I \in J_2 - J_1} c_l. \tag{1}$$

We suppose here that the tool management proceeds by first performing the removals, then moves and finally the insertions. The moves between two feasible storage states can then be done in two phases: by removing and inserting all the tools to be moved.

General two-level storage management problem (GSM-1). We are given $K, T \in \mathbb{N}$, $I \subset \mathbb{N}$, $c_a, c_r, c_m \in \mathbb{R}$, d and we want to find primary storage states $S(t)$, $t = 1, \ldots, T$, for which

$$\sum_{t=1,\ldots,T} c(S(t-1), S(t)) = \min \tag{2}$$

and the primary storage states are feasible with respect to the capacity K and tool requests d_t ($t = 1, \ldots, T$).

The two-level SM-problem without reorganization costs [13] can be seen as a special case of this problem by setting $c_a = c_r = 1$ and $c_m = 0$. This is because the primary storage can then always be reorganized without any cost. This also shows that the *GSM-1-problem is NP-hard* since the solution algorithm for it would also solve the SM-problem with zero cost reorganizations which is already known to be NP-hard [13]. One possible way to reorganize the primary storage is to defragment it every time when a tool is removed. By defragmentation we mean that all the tools are removed and placed sequentially to the primary storage starting from slot 1.

If we perform the reorganization in the way described in connection of Eq. (1) it is reasonable to set the values of the constants $c_a = c_r = 1$ and $c_m = 2$ due to the two-phase remove–insert operations described above.

3. Mathematical programming solution to GSM-1-problem

Let the set of tools be $I = [1, N]$. For all $t = 0, \dots, T$ and $i = 1, \dots, N$ we define the decision variables $x_{i,t}$ to be 1 if tool i is in the primary storage at time t, and 0 otherwise, i.e. for a particular PSS $S(t) = (J_t, \alpha_t)$:

$$x_{i,t} = \begin{cases} 1 & \text{if } x \in J_t, \\ 0 & \text{if } x \notin J_t. \end{cases}$$

We assume for the sake of simplicity that $c_a = c_r = 1$ and $c_m = 2c_a$. This turns the objective function (2) to the form:

$$\sum_{t=1,\dots,T} \sum_{i=1,\dots,N} c_i |x_{i,t} - x_{i,t-1}| = \min . \tag{2$\prime$}$$

Defining $y_{i,t} = |x_{i,t} - x_{i,t-1}|$, (2′) can be rewritten as

$$\sum_{t=1,\dots,T} \sum_{i=1,\dots,N} c_i y_{i,t} = \min . \tag{3}$$

The restrictions on the variables $x_{i,t}$ and $y_{i,t}$ are:

$$x_{i,t}, y_{i,t} \in \{0,1\} \quad (i = 1, \dots, N,\ t = 1, \dots, T), \tag{4a}$$

$$z_{i,t} \in \{1, \dots, K\} \quad (i = 1, \dots, N,\ t = 1, \dots, T), \tag{4b}$$

$$x_{i,0} = 0 \quad (i = 1, \dots, N), \tag{5}$$

$$x_{d_t,t} = 1 \quad (t = 1, \dots, T), \tag{6}$$

$$y_{i,t} \geqslant (x_{i,t} - x_{i,t-1}) \wedge y_{i,t} \geqslant -(x_{i,t} - x_{i,t-1}) \quad (i = 1, \dots, N,\ t = 1, \dots, T). \tag{7}$$

Eq. (5) states that we start from an empty primary storage state. Eq. (6) ensures that the tool needed by the present job is stored in the magazine and constraint (7) states that $y_{i,t}$ is the absolute value of $x_{i,t} - x_{i,t-1}$.

We still have to take care of the feasibility of the storage allocations. Let us therefore denote with $z_{i,t}$ the leftmost location index of tool i at time t. The location is relevant only if $x_{i,t} = 1$, too. The restrictions on the locations are

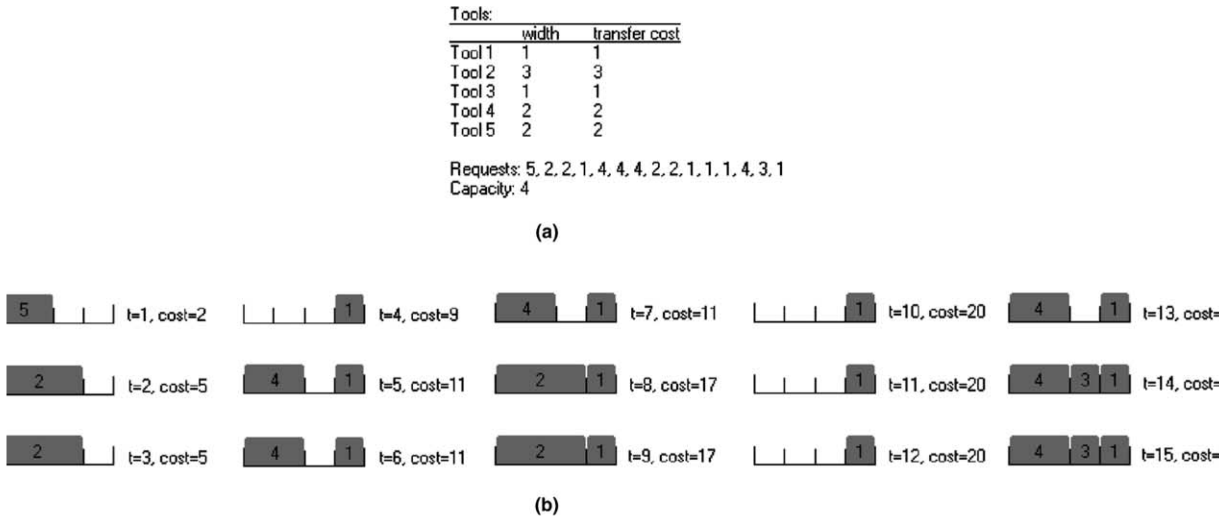
$$z_{i,t} + v_i \leqslant K \quad (i = 1, \dots, N,\ t = 1, \dots, T), \tag{8}$$

$$(x_{i,t} + x_{j,t} \leqslant 1) \vee (z_{i,t} + v_i \leqslant z_{j,t}) \vee (z_{j,t} + v_j \leqslant z_{i,t}) \quad (i,j = 1, \dots, N,\ i > j,\ t = 1, \dots, T). \tag{9}$$

Constraint (8) ensures that the right end of the tool magazine is not overridden. Constraint (9) prohibits overlapping of two tools at a given moment of time. The GSM-1-problem thus calls for minimizing (2′) subject to constraints (4)–(9).

3.1. An example

Consider a simple example where $|I| = 5$ and $T = 15$. Fig. 3a illustrates the problem setting. An optimal solution found by ILOG solver [19] is shown in Fig. 3b.


Tools:

	width	transfer cost
Tool 1	1	1
Tool 2	3	3
Tool 3	1	1
Tool 4	2	2
Tool 5	2	2

Requests: 5, 2, 2, 1, 4, 4, 4, 2, 2, 1, 1, 1, 4, 3, 1
Capacity: 4

(a)

Fig. 3. An example of the general SM-problem and its solution: (a) problem; (b) optimal solution.

4. Heuristic solution for GSM-1

Two efficient heuristics for the two-level SM-problem without reorganization costs are given by Matzliach and Tzur [13]. The first of these expresses the solution as a binary matrix where the rows stand for tools and the columns for part indices. The tools which are in the primary storage when processing a part are given by matrix elements of value 1. The number of tool switches can then be minimized by minimizing the number of zero blocks in the rows. This method can be seen as a global way of looking the problem, see also [4,13] in the context of tool switching problem.

The second algorithm is iterative and makes greedy decisions. It uses a generalization of the KTNS-rule by considering, in addition to the time elapsed until the next usage of tools, the transfer costs. An extra difficulty originates here from the fact that a particular tool to be inserted to the magazine may demand the removal of several smaller tools. One must therefore consider the costs of subsets of tools in the magazine and choose among these the most promising at each time point.

Both of the above algorithms could be modified to take into account the reorganization cost of the tool magazine. We propose here a heuristic based on the idea of the second algorithm by Matzliach and Tzur. A reason for this is that we can choose the subset of tools for removals in a rather straightforward way when the decisions deal with sequentially ordered tools instead all possible subsets of tools.

4.1. Tool distance metric

When inserting tools into the primary storage, one of the following three things can happen. First, there can be plenty of room for the new tool and we must choose where to place it. This is why we need a rule which chooses the place for each tool to be inserted. Secondly, there is only one possible set of free slots for the tool. In the third case, which is the most difficult one, the primary storage is full or does not have enough contiguous capacity for the new tool. In this case we must select one or more tools to be removed. In some cases the other possibility would be to reorganize the primary storage by moves of tools, but the move operation is too expensive to be useful in our cost model (1) because every move costs twice as much as remove ($c_m = 2$).

Let us define for all $i \in I$ and $t \in T$ the so-called *tool distance* dist(i, t). This value gives for tool i at time t the number of periods until it will be used the next time:

$$\text{dist}(i,t) = \begin{cases} m & \text{if } d_{t+m} = i \text{ and } d_{t+\tau} \neq i \text{ for } \tau = 1,\dots,m-1, \\ T & \text{if } d_{t+\tau} \neq i \text{ for } \tau = 1,\dots,T-t. \end{cases}$$

We can now define for any set of tools $J \subseteq I$ the weighted distance (called w-distance) to the next usage of the tools in J [13]:

$$w(J,t) = \frac{\sum_{i \in J} \text{dist}(i,t)/|J|}{\sum_{i \in J} c_i}.$$

4.2. Tool removal selection heuristic

We can use the w-distance to measure the attractiveness of different tool sets to be removed. All of these sets, however, are not suitable because we require that the tools to be removed free enough contiguous capacity for the tool we are about to insert. In particular, suppose that we are about to insert tool d_t and the primary storage state is $S(t-1) = (M, \alpha_1)$. Consider a tool set $J \subseteq M \backslash \{d_t\}$. Tools of J can be removed for making place for tool d_t if and only if there is a PSS $S(t) = (M \cup \{d_t\} - J, \alpha_2)$ which is feasible with respect to capacity K and $\alpha_1(i) = \alpha_2(i)$ for all $i \in M - J$. If this is true, we say that J is *valid for removal*. This means that in order to check the validity of a tool set we must find a injective mapping α_2 which gives a slot for every tool in the set $M \cup \{d_t\} - J$.

When selecting the tools to be removed we actually have a smaller number of options than in the model with zero reorganization costs: the tools should be sequential in the primary storage. This is because, if we would allow removals of tools which are not sequentially placed in the primary storage then we would be forced to make unnecessary removals and some reorganization with this strategy. Our heuristic for selecting a valid group of tools is given in Fig. 4. The primary routine *selectToolsToRemove* has four input parameters: the tool we are about to insert ($tool_a$), current primary storage state ($S = (J, \alpha)$), time index (t) and the size of the primary storage (K). The routine searches through all sequential tool sets of cardinality from 1 to $|J|$ and it returns the set $tools_r$ which is valid for removal and the w-distance is maximal. The auxiliary routine *isValidRemoval* tests whether or not the given tool set $tools_r$ is valid for removal when inserting $tool_a$ to S. This is done by forming a new storage state R, without $tools_r$. The routine *hasEmptySpaceFor* is then called for R and $tool_a$. The routine returns *true* if there is space for $tool_a$ in R, otherwise the result is *false* (note that this test is trivial in the problem without reorganization costs [13]).

In addition to the heuristics given above we need a rule for selecting the best position for the tool to be inserted. Several rules are known in the literature for this kind of problem [9]. The problem is to choose the best possible area from the free ones so that the future insertions can be handled with as little work as possible. We choose the space which is the smallest possible for the tool we are inserting (known as the Best Fit method). Two things can then happen: the size of the tool is the same as the size of the free space or the free space is larger than needed. In the latter case we must select the position of the tool within the chosen free space. We then simply place the tool at the beginning (i.e. left end) of the free area.

4.3. Time complexity analysis of the removal heuristic

In the worst case we have $|J| = K$. This happens when all the components in the PSS are of size 1 and the whole capacity is in use. This means that the three loops in the routine *selectToolsToRemove* consume time $O(K^3)$. In the innermost loop the tools in J are accessed in the order of the slot numbers. The sorting can be done before the main loop in an $O(K \log K)$ time. Routine *isValidForRemoval* is executed in the worst case

```
selectToolsToRemove($tool_a$,S,t,K) : set of tools          -- returns the most attractive tool set to
    best = {}                                                 -- be removed in the given situation
    let S = (J,$\alpha$)
    wb := 0                                                   -- the best w-distance value
    for size = 1 to |J|                                       -- search for subsets of size 1 to |J|
        for p = 1 to K
            $tools_r$ := {}
            for all $s \in J$ in increasing order of $\alpha(s)$
                if $\alpha(s) \geq p$ and $|tools_r|$ < size then
                    $tools_r := tools_r \cup \{s\}$
                end if
            end for

            wt := w($tools_r$, t)                             -- the w-distance of the current set
            if wb $\leq$ wt and isValidForRemoval($tools_r$, $tool_a$, J, $\alpha$) then
                best := $tools_r$
                wb := wt
            end if
        end for
    end for
    return best

isValidForRemoval($tools_r$, $tool_a$, J, $\alpha$) : boolean       -- returns true if the given set of tools is valid
    R := (J-$tools_r$, $\alpha$)                                    -- for removal
    return hasEmptySpaceFor($tool_a$, R)

hasEmptySpaceFor($tool_a$, S) : boolean                       -- returns true if there is empty space in the
    let S = (J,$\alpha$)                                          -- given magazine state for $tool_a$
    for p = 1 to K- $v_{tool_a}$ +1                             -- searching for a suitable location for $tool_a$
        empty := true
        for all $s \in J$                                     -- checking the overlapping for all tools in the
                                                              -- primary storage
            If ($p \geq \alpha(s)$ and $p < \alpha(s) + v_s$) or ($p + v_{tool_a} > \alpha(s)$ and $p + v_{tool_a} < \alpha(s) + v_s$) then
                empty := false
                break
            end if
        end for
        if empty then return true
    end for
    return false
```

Fig. 4. Tool removal heuristics.

K^2 times because of the number of the different sized sequences of tools. Each call demands $O(K^2)$ time. The time complexity of the selection routine is therefore $O(K^4)$. It can, however, be improved by using the following windowing system. Let us assume that we need an empty space of size e slots. Now we can search the space by moving a window of size e from slot 1 to the slot $K - e + 1$. On every slot we calculate the w-distance for the tools within the window and we select finally the window position which yields the largest w-distance. A tool is considered to be within the window if any of the slots is occupied by the tool. After we have calculated the best window position the routine returns the tool set it contains. Now, the number of different subsets of tools is at most K and therefore the overall complexity of this improved routine is $O(K^3)$.

4.4. Heuristic solution algorithm SMA for the GSM-1-problem

The algorithms given above can be collected to algorithm SMA which solves the GSM-1 problem heuristically. The algorithm proceeds iteratively through the tool requests. At each iteration $t = 1, \ldots, T$ it checks whether the tool d_t is already in the primary storage. If this is not true SMA first checks if there is empty space for the tool and inserts it if such a space is found. In the opposite case SMA removes a tool set by using the method described earlier. After the removal it inserts the new tool and advances t. Algorithm SMA (Fig. 5) has input parameters: tool set I, capacity K and tool requests d. The result is a sequence $S(1), \ldots, S(T)$ of primary storage states and the cost of this solution.

```
SMA(I, K, d) : cost                                                  -- solves the given
          Cost := 0                                                  -- GSM-1 instance
          S(0) := {}                                                 -- and returns the cost
          for t = 1 to T
                    let S(t-1) = (J, α)
          if $d_t \in I$ then S(t) := S(t-1)                              -- required tool is
                                                                     -- already in the
                                                                     -- primary storage
          else if hasEmptySpaceFor($d_t$, S(t-1)) then                   -- there is enough
                    S(t) := S(t-1) ∪ {(d(t), selectSlotForTool($d_t$, S(t-1))}   -- space for the tool
                    cost := cost + $c_{d_t}$
          else
                    $tools_r$ := selectToolsToRemove(tool, S(t-1), t)      -- freeing space for
                    α' := α                                          -- the tool
                    S(t) := S(t-1) ∪ {$d_t$ - $tools_r$, α')
                    α'(i) := selectSlotForTool($d_t$, S(t))
                    cost := cost + $\sum_{i \in tools_r} c_i + c_{d_t}$

          end if
end for
return cost
```

Fig. 5. SMA algorithm.

5. Experimental results for the one tool per job case

In the following test runs we evaluate the efficiency of SMA against the exact solutions. For this purpose we implemented the model of Section 3 (called OPT) for ILOG solver. It is, however, unlikely that ILOG is able to solve all the problem instances within reasonable time. Because of this, we use the exact solution method of the basic SM-problem (called LB) by Matzliach and Tzur [13] as a lower bound. We also use two naive algorithms (Naive and Random) as a benchmark. This enables us to evaluate the benefits gained by using more complex heuristic like SMA. Another purpose of these test runs is to estimate the contribution of the reorganization costs to the total switching cost. In the following test runs the cost factors are $c_a = 1$, $c_r = 1$ and $c_m = 2$. It is also possible in our test cases that the same tool appears several times in a row.

The Naive algorithm keeps only one tool at a time in the primary storage. This is why it is a kind of the worst case solving algorithm. The Random algorithm works iteratively by inserting tools to primary storage as they are needed. When the primary storage comes full it simply removes random tools from the storage until there is enough space for the tool we want to insert.

Problem set 1 uses the following parameters: $N = 25$ (number of tools), $T = 100$ (number of parts) and $K = 20, 50, 100, 150, 200$ and for Heuristic 2 $L = 3$ (maximum cardinality of the tool removal set, see [13]). Tool sizes are taken from uniform distribution of (7, 13). The problems are of proportional type which means that c_i/v_i is constant for all tools $i \in I$. The results are averaged over 30 problem instances. It turned out that all the problem instances were too hard to be solved optimally by ILOG. On the other hand, the lower bounds were easy to find. Statistical analysis with pairwise t-test shows that there is a very significant difference ($p = 0.01$) between the naive algorithms and SMA, see Table 2. The difference between SMA and the lower bound method (LB) is, however, not statistically significant for all capacities at this confidence level. Based on the results shown in Tables 1 and 2, SMA method is efficient in cost when compared against lower bound and naive methods. The running time for SMA is less than one second on Pentium 4 2.0 GHz for all problem instances in our test run. This is more than adequate for practical applications.

Problem set 2 uses the same parameters as set 1, but the transfer costs of the tools are all equal (=1). Note that the widths of the components are still different. This kind of situation is usual in the context of PCB assembly where widths of the component reels are not essential to the manual operation times of the component changes, but they make the feeder management more complicated. We observe from the results of Tables 3 and 4 that SMA finds solutions which are on the average only 3–12% worse than the lower bound. These bounds are almost identical to those of test run 1. Based on this, SMA works efficiently also when the transfer costs are not proportional.

Table 1
Summary of the results for the problem set 1 where $N = 25$, $T = 100$, tool costs are proportional to the widths and tool sizes are taken from uniform distribution of (7, 13)

Algorithm	Capacity									
	20		50		100		150		200	
	Avg	S.D.	Avg	S.D.	Avg	S.D.	Avg	S.D.	Avg	S.D.
SMA	1840	86.4	1270	94.9	782	89.6	503	75.5	331	54.1
Naive	1990	76.4	1990	76.4	1990	76.4	1990	76.4	1990	76.4
Random	1890	73.7	1660	96.6	1280	124	897	149	548	125
LB	1790	100	1160	84.5	700	78.7	453	63.6	309	41.8

In each cell we have the averaged costs of 30 problems instances. The running times of the heuristics varied between 0.1 and 0.5 seconds and the running time of LB varied between 10 and 600 seconds for a Pentium 4 2.0 GHz with Windows 2000 operating system.

Table 2
SMA algorithm is compared against the naive methods and lower bound method LBM

Comparison algorithm	Capacity									
	20		50		100		150		200	
	t-Test	Cost factor	*t*-Test	Cost factor	*t*-Test	Cost factor	*t*-Test	Cost factor	*t*-Test	Cost factor
Naive	*	0.925	*	0.638	*	0.393	*	0.233	*	0.156
Random	–	0.973	*	0.760	*	0.611	*	0.561	*	0.604
LBM	–	1.03	*	1.09	*	1.12	*	1.11	–	1.07

The left subcells show cases (marked by '*') where the differences are statistically very significant ($p = 0.01$, paired t-test). The right subcells give the average cost factor which is average SMA cost divided by average comparison algorithm cost.

Table 3
Summary of the results for the problem set 2 where $N = 25$, $T = 100$, tool costs are all equal (=1) and tool sizes are taken from uniform distribution of (7, 13)

Algorithm	Capacity									
	20		50		100		150		200	
	Avg	S.D.	Avg	S.D.	Avg	S.D.	Avg	S.D.	Avg	S.D.
SMA	181	4.25	124	6.48	77.1	6.36	49.6	6.50	32.7	4.14
Naive	199	0	199	0	199	0	199	0	199	0
Random	188	4.13	165	6.71	128	10.3	89.5	12.8	54.9	11.6
LB	175	5.00	115	5.65	69.0	587	44.5	5.07	31.0	3.37

In each cell we have the averaged costs of 30 problems instances. The running times of the heuristics varied between 0.1 and 0.5 seconds and the running time of LB varied between 10 and 600 seconds for a Pentium 4 2.0 GHz with Windows 2000 operating system.

Table 4
Statistical analysis of the problem set 2

Comparison algorithm	Capacity									
	20		50		100		150		200	
	t-Test	Cost factor	*t*-Test	Cost factor	*t*-Test	Cost factor	*t*-Test	Cost factor	*t*-Test	Cost factor
Naive	*	0.910	*	0.623	*	0.387	*	0.249	*	0.164
Random	*	0.963	*	1.33	*	0.602	*	0.554	*	0.596
LBM	*	1.03	*	1.08	*	1.12	*	1.11	–	1.05

SMA algorithm is compared against the naive methods and lower bound method LBM. The left subcells shows cases (marked by '*') where the differences are statistically very significant ($p = 0.01$, paired t-test). The right subcells give the average cost factor which is average SMA cost divided by average comparison algorithm cost.

6. The case of multi-tool requests

Generalizing the GSM-1-problem to a situation where *multiple tools are required during a time period* (called GSMM) makes the problem a lot harder to solve. The GSM-1-problem was relatively easy because we could remove all the tools necessary from the primary storage and bring a single tool to the empty space found in this way. When considering the case where a job uses multiple tools, one can meet a situation where simply removing tools is not sufficient. For this, consider the simple example in Fig. 6. Here, all the tools in the primary storage are required also during the next time period and we still need to insert a new tool which requires two slots. The only option is to reorganize the tools already in the primary stor-

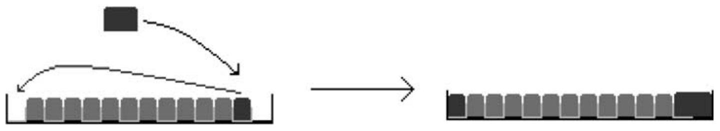


Fig. 6. Reorganizing the primary storage by moving the tools.

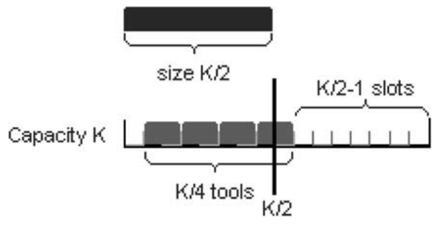


Fig. 7. Linear amount of moves required to reorganize the primary storage.

age. In order to do this, we need two kinds operations: moves and removes. Removes are needed when there is not enough empty space for a tool to be inserted. Moves are necessary when we are not able to remove enough tools from the primary storage because they are needed also during the current time period.

Another interesting fact is that *in the worst case scenario one needs a linear amount of moves* with respect to the capacity of the primary storage. To see this consider the case presented in Fig. 7. We have a primary storage with capacity K, and $K/4$ tools of size 2 located in slots $2, 4, 6, \dots, K/2$. All these tools are needed also during period t. We want to insert tool i of size $K/2$. Placing this tool into the primary storage requires $K/4$ moves, since all the tools in the primary storage must be moved.

We next propose two heuristics for the GSMM-problem. These algorithms use rather different ideas. The first one (SMMT-1) allocates a large contiguous area for the tools while the second one (SMMT-2) makes a distributed stepwise allocation.

6.1. Contiguous allocation, SMMT-1

Algorithm SMMT-1 starts by checking whether there is enough space for all the tools to be inserted. If this is true, they are simply inserted as in SMA. In the other case, we first remove the tools in d_t and then insert all the tools in d_t as a single aggregated "super-tool" of size $\sum_{i \in d_t} v_i$. This is done similarly as in SMA. Now, removing tools needed at step t corresponds to the rearrangement operation of the primary storage. The pseudocode for SMMT-1 is shown in Fig. 8. The parameters of the algorithm are like for the SMA-algorithm, except that d_t denotes now a set of tools. SMMT-1 uses subroutine smmtInsert, which is also used by the SMMT-2.

6.2. Distributed allocation, SMMT-2

The allocation of large contiguous areas from the primary storage may cause large transfer costs if the primary storage is fragmented. The problem may be avoided by inserting the tools of d_t one by one. We propose a heuristic SMMT-2 which keeps record of the tools we still need to insert. Let us suppose that toolset J will be inserted into the primary storage. The heuristics chooses randomly one tool i from J and inserts it in the primary storage. If i fits then another tool from J is taken and the process is iterated until all the tools have been inserted. When the tool does not fit in the primary storage, the w-distance

```
SMMT-1(I, K, d) : cost                              -- solves the given GSMM instance
    cost:=0                                         -- and returns the total cost
    for t=1 to T
        let S(t-1)=(J, α)
        if $d_t \subseteq J$ then                   -- the required tools are already in the
            S(t):=S(t-1)                            -- primary storage
        else if hasEmptySpaceFor($d_t \backslash J$) then   -- empty space for the missing tools
            J'=J                                    -- is found
            $\alpha_t=\alpha$
            S(t)=(J', $\alpha_t$)
            for all $i \in d_t \backslash J$        -- inserting the required tools one by one
                $\alpha_t$(i):=selectSlotForTool(i,S(t))
                J':=J'$\cup${i}
                cost:=cost+$c_i$
            end for
        else                                        -- there is not enough space for the tools
            S(t), $cost_a$:=smmtInsert(S(t-1),t)    -- we need to insert
            cost:=cost+$cost_a$
        end if
    end for
    return cost

smmtInsert(S, t): storage state, cost               -- inserts the tools required as a one "super-tool"
    let S=(J, α)
    $\alpha_t$:=α
    J':=J \ $d_t$                                   -- creating a new primary storage state with
    S':=(J', $\alpha_t$)                            -- none of the tools required during t
    cost:=cost+c(S,S')                              -- adding the costs of removing tools in $J \cap d_t$

    i:=new tool                                     -- creating a super tool
    $v_i$:=$\sum_{i \in d_t} v_i$
    $c_i$:=$v_i$

    $tools_r$:=selectToolsToRemove(i,S,t)           -- freeing space for the super tool
    J':=J'-$tools_r$
    for all $i \in d_t$                             -- inserting the required tools
        $\alpha_t$(i):=selectSlotForTool(i,S')      -- one by one
        J':=J' $\cup$ {i}
    end for
    cost:=$\sum_{i \in tools_r} c_i + \sum_{j \in d_j} c_j$   -- adding the costs of tools in $d_t$

    return S', cost
```

Fig. 8. Pseudocode of SMMT-1 algorithm.

```
SMMT-2(I, K, d) : cost                              -- solves the given GSMM instance
    cost:=0
    for t=1 to T
        let S(t-1)=(J,α)
        if $d_t \subseteq J$ then
            S(t):=S(t-1)                            -- all required tools are already
        else if hasEmptySpaceFor($d_t \setminus J$) then   -- in the primary storage
            J':=J                                   -- there is enough empty space
            $α_t$:=α                                -- for all the tools
            S(t):=(J',$α_t$)
            for all $i \in d_t \setminus J$
                $α_t$(i):=selectSlotForTool(i,S(t))
                J':=J' ∪ {i}
                cost:=cost+$c_i$
            end for
        else
            J':=$d_t \setminus J$                   -- the tools we need to insert require
            R:=S(t-1)                               -- more empty space than there is
            let R=(L,$α_r$)                         -- currently available set J' is the set of
            cyclecost:=0                            -- tools we still need to insert
            S', $cost_{smmt}$:=smmtInsert(S(t-1),t)
            while $J' \neq \emptyset$ and cyclecost≤$cost_{smmt}$     -- trying to insert all
                i:=random tool in J'                          -- the tools in $d_t$
                if hasEmptySpaceFor(i,R) then
                    L:=L ∪ {i}
                    $α_r$(i):=selectSlotForTool(i,R)
                    cyclecost:=cyclecost+$c_i$
                else
                    $tools_r$:=selectToolsToRemove(i,R,t)
                    J':=J' ∪ ($d_t \cap tools_r$)             -- tools which are
                    L:=L - $tools_r$                          -- in d(t) are merged
                                                              -- to J'
                    cyclecost:=cyclecost+c($tools_r$)
                    L:=L ∪ {i}
                    $α_r$(i):=selectSlotForTool(i,R)
                    cyclecost:=cyclecost+$c_i$
                end if
            end while
            if      cyclecost>smmtcost then                   -- if smmt-2
                    S(t):=S'                                  -- is too expensive
                    cost:=cost+$cost_{smmt}$                  -- then use smmt
            else

                    cost:=cost+cyclecost
                    S(t):=R
            end if
            end if
        end for
        return cost
```

Fig. 9. Pseudocode of SMMT-2 algorithm.

metric is used to choose a set of tools $tools_r$ to be removed from the primary storage. The tools in $tools_r$ which are not needed during this time period ($tools_r \backslash d_t$) are sent to the secondary storage. Other tools which are needed during the current time period, $tools_r \cap d_t$, are merged to set J so that they will be inserted later. In this way, the algorithm reorganizes some parts of the primary storage.

SMMT-2 has the potential defect of creating an infinite loop of insertions and removals. These kinds of cycles can be detected by remembering all the previous (PSS, J)-pairs. Even this is not necessary since we can always switch to the heuristic used by SMMT-1 when the cost of the insertions and removals becomes larger than the cost of the super-tool heuristic. This is the method the actual implementation of SMMT-2 uses. For the pseudocode of SMMT-2, see Fig. 9.

One might think that the tools which are needed during the current time period and are already in primary storage would need a larger weight because when removing them they must also be reinserted. However, this is not necessarily the case because the w-distance of these tools is zero and their costs thus reduce the w-distance.

7. Experimental results for the multi-tool case

The following set of tests measures the performance of the heuristics described in Section 6. The solutions found by SMMT-1 and SMMT-2 are evaluated against each other to find out whether the added complexity of the SMMT-2 pays off. The second objective is to measure them against the lower bounds found by using the model of Matzliach and Tzur [13] (called LBM). The results are also evaluated against naive methods (Naive and Random, see Section 5) to confirm that more elaborate heuristics give significantly better results and are therefore indispensable. Differences are analysed with paired t-test.

Problem set 3 was created with parameters $N = 25$, $T = 40$ and $K = 60, 100, 150, 200$. Tool sizes were taken from the uniform distribution of (7, 13). The transfer costs for all tools are equal to their sizes. Tool set cardinalities for each time period were taken from the uniform distribution $(1, 4)$. Each tool had an equal probability to be chosen in each of these tool sets. The results were averaged over 30 randomly generated problem instances. Table 5 shows a summary of the test results. We observe that SMMT-2 outperforms clearly SMMT-1. The difference is rather large for large capacities and SMMT-2 compares surprisingly well even with the lower bounds found by LBM. The running times for all heuristics were between 0.1 and 0.6 seconds and the running time of LBM implemented with ILOG solver varied from 15 to 650 seconds. The results are slightly worse for SMMT-2 when compared against the LBM than in the sin-

Table 5
Summary of the results for problem set 3 where $N = 25$, $T = 40$, $K = 60$, 100, 150, 200 tool sizes are uniformly distributed from (7, 13), transfer costs are proportional and the tool set cardinality is from uniform distribution of $(1, 4)$

Algorithm	Capacity							
	80		120		160		200	
	Avg	S.D.	Avg	S.D.	Avg	S.D.	Avg	S.D.
SMMT-1	4320	363	3400	338	2440	365	1480	265
SMMT-2	2700	277	1760	200	1130	156	687	132
Naive	5070	286	5070	286	5070	286	5070	286
Random	3850	360	3090	363	2310	284	1520	257
LBM	2340	254	1490	182	942	132	560	99.3

The cells of the table show the average cost over 30 problem instances and the standard deviation of the costs. The running times of the heuristics varied between 0.2 and 1.0 seconds and between 15 and 650 seconds for the LBM when using Pentium 4 2.0 GHz with Windows 2000 operating system.

Table 6
Summary of the results for problem set 4 where $N = 25$, $T = 40$, $K = 60$, 100, 150, 200 tool sizes are uniformly distributed from (7, 13), transfer costs are all equal (=1) and the tool set cardinality is from uniform distribution of (1, 4)

Algorithm	Capacity							
	80		120		160		200	
	Avg	S.D.	Avg	S.D.	Avg	S.D.	Avg	S.D.
SMMT-1	426	30.1	335	26.6	242	11.6	147	28.5
SMMT-2	261	20.1	171	16.8	111	11.6	66.4	11.0
Naive	503	19.9	503	19.9	503	19.9	503	19.9
Random	382	27.8	305	28.8	228	22.4	151	20.4
LBM	232	18.5	146	14.1	92.3	10.4	55.1	7.67

The cells of the table show the average cost over 30 problem instances and the standard deviation of the costs. The running times of the heuristics varied between 0.2 and 1.0 seconds and between 15 and 650 seconds for the LBM when using Pentium 4 2.0 GHz with Windows 2000 operating system.

gle-tool case when comparing SMA and LBM. The differences between SMMT-2 and all other methods in Table 5 were statistically very significant ($p = 0.01$) for all cases.

The problem set 4 uses the same parameters as set 3, but the transfer costs of the tools are all equal (=1) although the widths of the components are still different, cf. problem set 2. We observe from the results of Table 6 that SMMT-2 finds solutions which are on the average only 13–21% worse than the lower bounds. These bounds are almost identical to those of test run 1. Again, all differences for SMMT-2 are statistically very significant. Based on this, SMMT-2 works efficiently also when the transfer costs are not proportional.

8. Conclusions

The general two-level SM-problem with a single tool per part (GSM-1) and its multi-tool extension (GSMM) were considered. It was assumed that each tool needs a contiguous area from the primary storage. Another assumption was that one cannot defragment the primary storage without the cost of moving tools. In the mathematical optimization model it was assumed that moves are performed as a two-phase action of remove–insert operations. This is a pessimistic view due to the possibility that in some cases it is possible to use chains of moves directly to the new places. But on the other hand, these kind of operations are hardly realistic in practice.

Three heuristics and an integer programming method were proposed to solve the GSM-1 and GSMM problems. The numerical results with integer programs showed clearly the need for heuristic solutions. Optimal solutions took a very long time to calculate even for small problem instances. The optimal solution of GSM-1 could not be found for any of the problems with 100 tool requests within the time limit of three days. The simple SM-variants of this size were solved easily, however.

The numerical results show that the heuristic solutions are on the average quite close to the optimal solutions in the test cases. The algorithms are very fast for problems of practical size. The new algorithms take advantage of the fact that in the GSM problem we have a restricted number of options when selecting the tool set to be removed: one has only to consider those subsets of tools which are sequentially ordered in the primary storage. This is sufficient if the move cost $c_m \approx c_a + c_r$. If however, the move cost is much smaller than the joint cost for insertion and removal, one should allow that the tools to be removed are scattered. Then, the new tool may fit the free space better and some greater tools which are needed soon may stay in the primary storage.

Two heuristics for the multi-tool case were proposed: SMMT-1 and SMMT-2. SMMT-1 uses the super-tool heuristics when inserting tools. SMMT-2 uses a method where the tools are inserted one by one, but it also uses the super-tool heuristics as a backup strategy when the iterative insertion procedure would cost more than the super-tool heuristics. It turned out that the iterative method saves much work.

The research on the tool switching and job grouping [10] have omitted the magazine reorganization costs. The methods given in this work are readily applicable in these two, otherwise well studied problems. One can instead of the (approximative) KTNS-rule use now the more realistic multitool heuristics. Another topic of research deals the organization of the magazine. A linear tool magazine was assumed as is common in PCB assembly machines. The situation changes slightly if the magazine is circular so that the first slot follows physically the last one.

References

[1] M.S. Akturk, J.B. Ghosh, E.D. Gunes, Scheduling with tool changes to minimize total completion time: A study of heuristics and their performance, Naval Research Logistics 50 (2003) 15–30.

[2] S. Avci, M.S. Akturk, Tool magazine arrangement and operations sequencing on CNC machines, Computers Operation Research 23 (11) (1996) 1069–1081.

[3] Y. Crama, J. van de Klundert, The approximability of tool management problems, Technical Report rm96034, Maastricht Economic Research School on Technology and Organisation, 1996.

[4] Y. Crama, A.W.J. Kolen, A.G. Oerlemans, F.C.R. Spieksma, Minimizing the number of tool switches on a flexible machine, The International Journal of Flexible Manufacturing Systems 6 (1994) 33–53.

[5] Y. Crama, J. van de Klundert, F.C.R. Spieksma, Production planning problems in printed circuit board assembly, Working Paper GEMME 9925, University de Liege, 1999.

[6] H. Djellab, K. Djellab, M. Gourgand, A new heuristic based on a hypergraph representation for the tool switching problem, International Journal of Production Economics 64 (2000) 165–176.

[7] M. Hirvikorpi, K. Salonen, T. Knuutila, O.S. Nevalainen, General two level storage management problem-reconsideration of the KTNS-rule, Technical Report 532, TUCS-Turku Centre for Computer Science, 2003.

[8] M. Johnsson, Operational and tactical level optimization in printed circuit board assembly, PhD thesis, Turku University, 1999.

[9] D.E. Knuth, The Art of Computer Programming, vol. 1, 3rd Ed., pp. 435–456, 1998.

[10] T. Knuutila, O. Nevalainen, A reduction technique for weighted grouping problems, European Journal of Operational Research 140 (2002) 590–605.

[11] V.J. Leon, A. Peters, A comparison of setup strategies for printed circuit board assembly, Computers & Industrial Engineering 34 (1) (1998) 219–234.

[12] B. Matzliach, M. Tzur, The online tool switching problem with non-uniform tool size, International Journal of Production Research 36 (12) (1998) 3407–3420.

[13] B. Matzliach, M. Tzur, Storage management of items in two levels of availability, European Journal of Operational Research 121 (2000) 363–379.

[14] V.S. Mookerjee, Policies for data archival in hierarchical storage management, European Journal of Operational Research 138 (2002) 413–435.

[15] A.S. Tanenbaum, Modern Operating Systems, second ed., Prentice-Hall, 2001, pp. 200–201.

[16] C.S. Tang, E.V. Denardo, Models arising from a flexible manufacturing machine, part I: Minimization of the number of tool switches, Operations Research 36 (5) (1988) 767–777.

[17] A. Turkcan, M.S. Akturk, R.H. Storer, Nonidentical parallel CNC machine scheduling, International Journal of Production Research 41 (10) (2003) 2143–2168.

[18] M. Tzur, A. Altman, Minimization of tool switches for a flexible manufacturing machine with slot assignment of different tool sizes, IIE Transactions 36 (2004) 95–110.

[19] <http://www.ilog.com/products/solver/>, ILOG Solver.

Publication IV

Hirvikorpi M., Johnsson M., Knuutila T. and Nevalainen O.S., A General Approach to Grouping of PCB Assembly Jobs. To appear in *International Journal of Computer Integrated Manufacturing*, 2004.

A General Approach to Grouping of PCB Assembly Jobs

Mika Hirvikorpi[2], Timo Knuutila[2], Mika Johnsson[1] and Olli S. Nevalainen[2].

[1]Valor Computerized Systems OY, Ruukinkatu 2, 20540 Turku

[2]Turku University, Department of Information Technology and TUCS
Lemminkäisenkatu 14 A, 20520 Turku

Abstract

Ordering of batches of printed circuit boards (PCB) has a significant impact to the efficiency of the electronic component placement processes. Through PCB batch grouping we aim to minimize the total setup time between batches. Batch groups are formed so that each group can be handled with one component setup. The job grouping problem calls for a set of groups with minimal cardinality.

This paper considers three variants of the job grouping problem under a single formulation. A generic mathematical formulation which solves all three of them is given. The variants are abstractions of machine organizations used in present day manufacturing systems. Inspired by the new formulation of the exact solution method a joint heuristics suited for all the three problems is also constructed. Benefits of the new algorithm are its conceptual simplicity, adaptivity to different machine layouts and competitive computational efficiency.

Keywords: Printed circuit board, component placement, electronics manufacturing, surface mounted components, flexible machines

1. Introduction

One of the most important processes in electronic industry is the placement of electronic components on printed circuit boards (PCBs). This is done by using high precision automated placement machines. When working with only one placement machine, there are several different ways to organize the component insertions (Leon and Peters 1998):

- every batch of identical PCBs can be handled separately,
- the order in which the PCBs are processed can be selected so that the overall setup time is minimized, and
- the number of setup occasions can be minimized by grouping the PCB batches.

The first organization is common for 'low mix – high volume' production, in which one has only few different PCB types to manufacture but the production volumes are very large. In this type of production, most of the time is spent on the actual placement of the components and optimization is more fruitful on the level of a single PCB type. Optimization on this level is called operational, see Johnsson 1999 for the categorization. One of the main issues on the operational level is the optimization of the movements of the printing head and this way minimizing the printing time.

Selecting the processing order of PCBs properly is beneficial when the production volumes are medium sized and the overall time used for tool switches is substantial. The problem here is to find such a processing order and component management decisions for the batches that the total number of component switches is minimized. The component switches are made between different types of PCB batches because the feeder (holder of the components) of the component placement machine is of limited capacity. This approach assumes that each component switch requires certain fixed time, so the overall time required for the component switches is simply the sum of the individual switching times. The calculation of exact solutions for the tool switching problem are possible only for small problem instances because it is NP-hard, as shown by Crama *et al.* 1994. Several efficient heuristics have been proposed for the problem, see Van Hop 2003, Hertz *at al.* 1998, Bard 1988 and Al-Fawzan and Al-Sultan 2003. Study of Al-Fawzan and Al-Sultan 2003 considers a new heuristic approach for the tool switching problem. Shirazi and Frizelle 2001 study the current state of industry in applying the methods developed for the tool switching problem to practice.

The third approach of PCB component placement assumes fixed time setup events, *i.e.* component loading is either based on a full tear-down or on the use of several changeable feeders. Therefore, by minimizing the number of setup events one minimizes the total time used for the component setups. The *job grouping problem* (JGP) asks for a feasible minimal cardinality grouping of the PCB batches. A group of PCB batches is said to be feasible if all the components required by the batches within the group fit in the feeder simultaneously. After the groups have been formed, each group can be processed without any interruption. Grouping of jobs is favorable especially when the number of different PCB types is large and the production volumes are small. This approach is actually an application of the group technology (GT) which is applied widely in the field of flexible manufacturing systems.

The present study concentrates to the job grouping problem, which has significant impact on the production costs of medium and small volume producers. The JGP is well known and it has been studied extensively in the past, see Carmon *et al.* 1989, Hashiba and Chang 1992 and Maimon and Shtub 1991. The flexible component placement machines using new technological solutions of arranging the component feeder require new novel algorithms. In particular, we study here three different types of component placement machines and describe their impact on the formulation and solution of the grouping problem.

The basic JGP assumes a linear feeder unit which is composed of a certain number of slots to store the component reels. It has been assumed in most reports that each component reel takes one slot of the feeder capacity. However, this assumption rarely holds in practice nowadays – there are usually at least 3-4 different widths of component reels. A review of different approaches to solving the JGP is given by Knuutila *et al.* 2001. These approaches include exact solving by using logic and mathematical programming. They also introduced several heuristics which gave near optimal results in practical tests. The approximability of the JGP has been studied by Crama and Klundert 1996. They also show that JGP is NP-hard.

The second variant of the JGP (called JGP-T) assumes that the placement machine can handle several different types of feeders at the same time (*c.f.* Universal GSMs). For example there can be separate areas for tape feeders, tray feeders and tube feeders. Because all the traditional algorithms for solving the JGP assume only one linear feeder unit with certain capacity, they clearly cannot handle this type of feeder organization. Knuutila *et al.* 2004 showed that this problem is NP-hard, presented an exact solution through mathematical programming and developed heuristic solution algorithms for it.

The JGP-B (Hirvikorpi *et al.* 2004), which is the third variant of JGP, assumes a machine which uses separate feeder boxes and the feeder unit itself is only a holder for these boxes. There are usually several different types of boxes in use and each box type has different limitations for the component types it can accommodate. This makes the problem a lot harder, because, in addition to searching the grouping, one must also find a suitable feeder configuration for each group.

Until now these three problems have been considered separately by using algorithms specially suited for the particular problem variant in question. In the present study it is demonstrated how all of them can be formulated as a special case of JGP-B. Although this is a rather simple technical matter, it rises a more important question dealing the applicability of the JGP-B algorithms for the simpler problems. When trying to evaluate the practical usefulness we took another look at the solution algorithms of JPG-B and reorganized them by applying the most successful ideas of the effective JGP and JGP-T heuristics, see Maimon and Shtub 1991 and Knuutila *et al*. 2001. As a basis of this, a new JGP-B heuristic is introduced. The algorithm is profiled for problem instances used in previous studies for all the three problems.

The rest of the presentation is organized as follows. Section 2 defines the different variants of the JGP mathematically. Section 3 presents a MILP formulation which is able to solve all the problems of section 2. Section 4 introduces problem variant independent heuristic algorithms for the JGP. In order to keep the representation self-contained we recall shortly the methods used for the standard JGP and JGP-T. The efficiency of the new heuristic and exact solution methods is profiled and compared to previous methods in section 5. The paper is concluded in section 6.

2. Definitions

The variants of the JGP have several common concepts which are defined first. The difference of the problem settings lies in the organization of the feeder unit. Because of this it is justified to divide the definitions into two parts. The first part consists of the component and PCB definitions and the second part deals with the feeder organization. The following formulations do not directly follow any mathematical formulation presented earlier for the JGP, instead see Knuutila *et al.* 2001, Knuutila *et al.* 2004 and Hirvikorpi *et al.* 2004 for specialized definitions of JGP, JGP-T and JGP-B.

In all the three job grouping problems one is given a set of component types E and a set of PCB types C. Each PCB type is considered as a collection of component types. The technology of placement machines enables us to handle the PCB types as *sets* of component types, *i.e.* in JGP one does not worry about the actual number of occurrences of each component type on the PCBs. This is due to the fact that the feeder of the component placement machine is limited with respect to the number of different component types, but the supply of a certain component type is considered to be infinite. The reason for this assumption is that the component packages contain a large number of identical components and individual packages can be changed quickly when needed. The PCB types and component types are defined formally as follows:

W	set of widths of component types $W \subset \mathbb{N}$,
E	set of component types, each component type $e \in E$ has the width property $w(e) \in W$, and for each subset E' of E the following properties are defined:
	$w(E')$ — the sum of widths for component types in E',
	$w(E',w')$ — the sum of widths for component types of width w' in E',
C	set of PCB types, for each PCB type $c \in C$ it holds: $c \subseteq E$,
G	a set of groups of PCB types, each group $g \in G$ has the following properties:
	$w(g)$ — the sum of widths of the components in g, and
	$E(g)$ — set of all components in g, *i.e.* the set $\cup_{c\in g}\, c$.

2.1. The basic JGP

The traditional feeder organization in component placement machines is the linear feeder, which consists of sequentially ordered slots for storing the components. The number of slots in the feeder (K) is called the capacity of the feeder. Depending on the actual widths of the components placed in the feeder, it can accommodate up to K different component types. Each component type $e \in E$ allocates $w(e)$ (≥ 1) slots from the feeder.

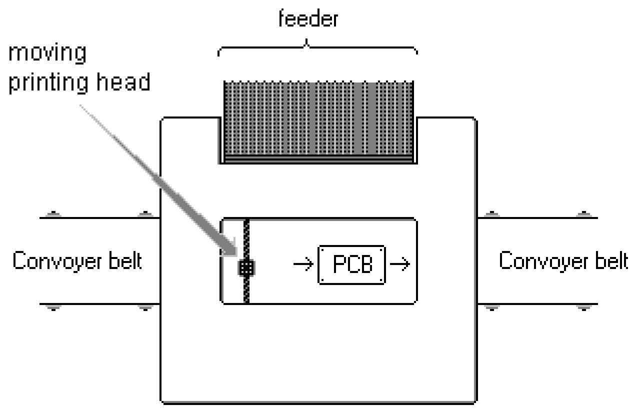


Figure 1. A component placement machine with a linear component feeder.

Figure 1 demonstrates the working principle of a component placement machine with a linear feeder. The bare PCBs enter the machine on the conveyer belt from the left. Inside the machine there is a moving printing head which places the components on the PCB. Depending on the type of the printing head it can retrieve one or several components from the feeder at the same time. In some machines the position of the feeder is fixed and the printing head retrieves the components itself while in others the feeder moves.

It is assumed in the job grouping problem that the whole feeder is either replaced with another (by changing removable feeder units or so-called feeder banks) or totally reloaded when advancing to the assembling of the next batch of PCBs. This is why the setup event between PCB batches takes a fixed time and thus the number of setup events is the factor to be minimized. As stated above, the machine operator adds more components to the feeder during the processing if some component type is about run out. It is however not easy to change the component types since it would require changing the machine control program which guides the printing.

A grouping G is said to be *feasible* with respect to a given capacity K if and only if for all $g \in G$ it holds $w(g) \leq K$.

We can now define the basic *Job grouping problem with a linear feeder* (JGP): given a set of PCB types C, a set of component types E and capacity K, find a feasible grouping G with minimal cardinality.

2.2 JGP-T

While the common JGP is restricted to the use of a single feeder, new technological advances have removed this limitation, and some machines, like Universal GSM-1 (Knuutila *et al.* 2004), are able to use several techniques simultaneously. In practice this is implemented so that the machine has several feeders each of which has a certain capacity. Figure 2 shows the working principle of a component placement machine of this kind. In contrast to a machine equipped with one linear feeder the printing head now retrieves components from several different stations according to the feeder type of the component. Otherwise the working principle is the same.

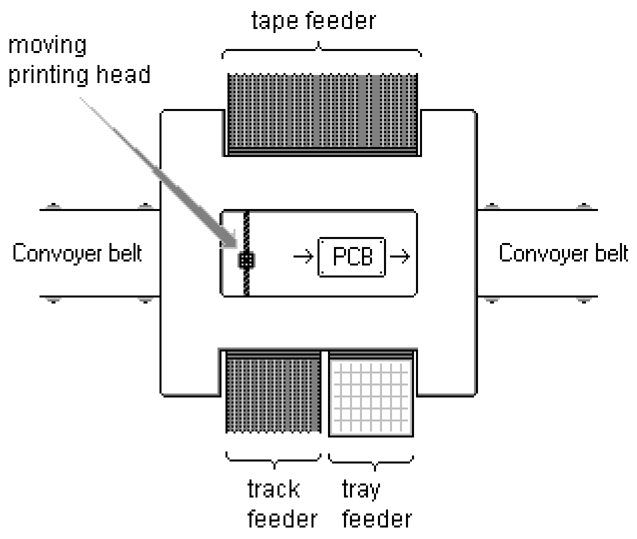


Figure 2. A component placement machine with multiple feeders.

In addition to the width, each component type $e \in E$ has now also the feeder type $t(e) \in T$, where $T \subset$ N is the set of all feeder types. The function w is extended as follows:

$w(E',t')$	the sum of widths of component types $E' \subseteq E$ for feeder type t'.

The feeder configuration F is now described by type dependent capacities:

$F(t')$	the capacity of type t' feeder.

Grouping G is said to be *t-feasible* with respect to a given feeder configuration F and the feeder type set T if and only if for all $g \in G$ and $t' \in T$ holds: $w(E(g),t') \leq F(t')$.

Job grouping problem with feeder types (JGP-T): given a set of PCB types C, set of component types E and feeder configuration F, find a t-feasible grouping G of minimal cardinality.

2.3 JGP-B

The job grouping problem can be generalized further by increasing the flexibility of the placement machine. Some of the most recent machines (*e.g.* MyData), use a feeder unit arranged to a set of replaceable boxes. These belong to two categories, *simple* and *flexible*, according to the restrictions on the component types they can hold.

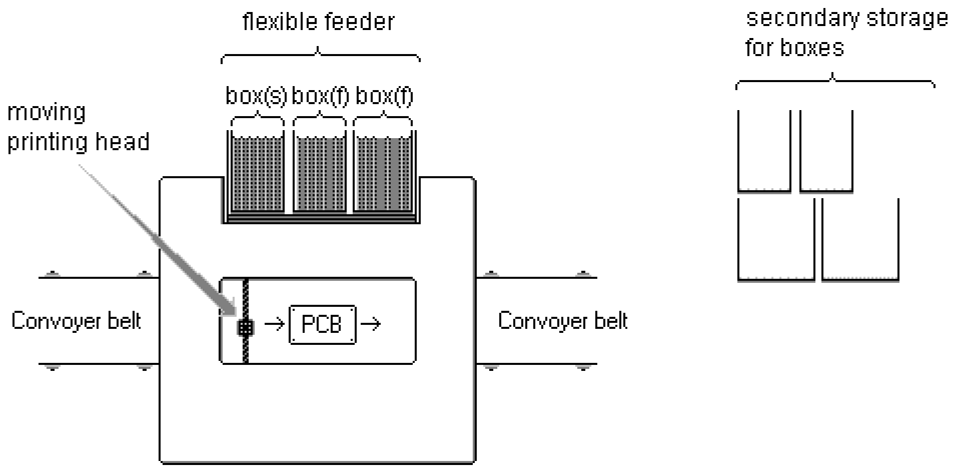


Figure 3. A component placement machine with a flexible feeder unit.

Simple boxes can hold component types of a certain single width only. A simple box has a fixed capacity which indicates how many component types it can store simultaneously. A flexible box can store components of several different widths at the same time, but a component type may consume more slots than its nominal width would indicate in these boxes. This is the cost paid from the extra flexibility. For example if we have a component of width two it might occupy three slots in a flexible box. The situation is further complicated by the fact that there are usually more boxes available than the feeder can hold simultaneously. The number of boxes the feeder can hold depends on the external width of the boxes. This is because the feeder itself has certain maximal width. Figure 3 shows an example of this type of machine. The feeder is now simply a holder for the boxes.

The job grouping problem with a flexible feeder requires the following notations:

B	set of boxes, each $b \in B$ has the following properties:	
	$w(b)$	the external width of box b,
	$K(b)$	the capacity of box b,
	$u(b,e)$	number of slots the component type e uses from box b,
	$a(b)$	set of allowed component type widths (for simple boxes $\|a(b)\|=1$) in b.
F	flexible feeder unit:	
	$w(F)$	width of the feeder unit F,
	$n(F)$	maximum number of boxes the feeder F can hold simultaneously.

Mapping function m:$E' \rightarrow B'$ is said to be feasible with respect to a set of boxes $B' \subseteq B$ and a set of component types E' if :

1) $m(e) \in B'$ for all $e \in E'$,
2) $w(e) \in a(m(e))$ for all $b \in B'$ and $e \in E'$,
3) $\sum_{m(e) \in b} u(b,e) \leq K(b)$ for all $b \in B'$.

Group g is said to be feasible with respect to a set of boxes B' if there exists a mapping function m which is feasible with respect to B' and $E(g)$.

Box set $B' \subseteq B$ is feasible with respect to feeder F if $|B'| \leq n(F)$ and $\sum_{b \in B'} w(b) \leq w(F)$.

Grouping G is *b-feasible* with respect to a box set B and feeder unit F, if for all groups $g \in G$, there exists a feasible box set $B' \subseteq B$ which is feasible with respect to F.

Job grouping problem for flexible feeder (JGP-B): when given a set of PCB types C, a set of component types E, a set of boxes B and feeder unit F, find a b-feasible grouping G with minimal cardinality.

3. Exact solution

The studies Knuutila *et al.* 2001, Knuutila *et al.* 2004 and Hirvikorpi *et al.* 2004 present mathematical programming models specific for the three job grouping problems of the previous section. We next give a generic model which can be specialized to the variants in a simple and efficient way. It is our hypothesis that the generalized model is able to solve problem instances of roughly the same size for all these problem variants as the previous problem specific models. This hypothesis is tested in section 5. The parameters of the new JGP-G model are as follows (i=1,…,|E|, k=1,…,|B|, j=1,…,|C|):

e_{ik}	the number of slots consumed by component type i from box k,
c_{ij}	1 if PCB type j requires the component type i, 0 otherwise,
c_j	the number of component types in PCB type j,
b_k	the width of box k,
d_k	the capacity of box k,

N	the maximum number of boxes the feeder can hold,
W	the width of the feeder, and
U	an upper bound for the number of groups ($U \le |C|$).

The decision variables are (l=1,…,U):

x_{jl}	1 if PCB type j is in group l, 0 otherwise,
y_{kl}	1 if box k is in the feeder configuration of group l, 0 otherwise,
z_l	1 if group l is not empty, 0 otherwise,
m_{ikl}	1 if component type i is in box k of the feeder configuration of group l, 0 otherwise.

The general job grouping problem (JGP-G) can be defined as follows:

minimize

$$\sum_{l=1,\ldots,U} z_l \qquad (1)$$

subject to

$$\sum_{j=1,\ldots,|C|} x_{jl} \le z_l \,|\, C \,| \qquad \text{for all } l=1,\ldots,U \qquad (2)$$

$$\sum_{i=1,\ldots,|E|} (c_{ij} (\sum_{k=1,\ldots,|B|} m_{ikl})) \ge c_j x_{jl} \qquad \text{for all } l=1,\ldots,U, j=1,\ldots,|C|, \qquad (3)$$

$$\sum_{k=1,\ldots,|B|} y_{kl} \le N \qquad \text{for all } l=1,\ldots,U \qquad (4)$$

$$\sum_{k=1,\ldots,|B|} y_{kl} b_k \le W \qquad \text{for all } l=1,\ldots,U \qquad (5)$$

$$\sum_{i=1,\ldots,|E|} e_{ik} m_{ikl} \le d_k \qquad \text{for all } l=1,\ldots,U, k=1,\ldots,|B| \qquad (6a)$$

$$\sum_{k'=1,\ldots,|B|} m_{ik'l} \le y_{kl} \qquad \text{for all } i=1,\ldots,|E|, k=1,\ldots,|B|, l=1,\ldots,U \qquad (6b)$$

$$\sum_{l=1,\ldots,U} x_{jl} \ge 1 \qquad \text{for all } j=1,\ldots,|C|. \qquad (7)$$

Formula (1) gives the number of groups which is to be minimized. An upper bound for the number of groups U can be obtained from a heuristic algorithm, for example. Constraint (2) says that a group is not empty if any of the PCB types is assigned to it. Constraint (3) demands that if a PCB type is assigned to a certain group then all its component types are mapped to the feeder configuration of that group. Constraints (4) and (5) state that the box sets of the groups are suitable for the feeder with respect to the number of boxes N and their total width W. Constraint (6a) quarantees that the capacities of the boxes are not exceeded and constraint (6b) prevents from using boxes which are not in the configuration of the group. It also states that component can be set to one box in a group, only. Constraint (7) requires that each PCB type is assigned to at least one group.

The basic JGP can be solved with this model by transforming the feeder to a single flexible box with a given capacity C. JGP-T requires the transformation of each feeder type to a flexible box. In addition the parameter e_{ik} must force the constraints concerning feeder types. Assume that the feeder type of component type i is k, then the parameter e_{ik} is set to the width of the component type i. For other boxes it is set to d_k+1 which makes it impossible to put the component to any other box. Similarly, for a JGP-B instance the e_{ik} is set to d_k+1 if the box is unsuitable to hold the width of component type i.

4. Heuristics

Even the basic JGP is algorithmically hard (*i.e.* it is NP-hard, see Crama and Oerlemans 1994). Solving it optimally is not possible within reasonable time for large number of PCBs. Even its accurate approximation has been proved to be hard (Crama and Klundert 1996). With the latest integer and logic programming software one can (with good luck) solve problems of size 30 PCBs optimally, as reported in Knuutila *et al.* 2001. The advantage of using logic programming is that one can easily add case-specific constraints like ordering relations, which increase the flexibility of the solution method, see Knuutila *et al.* 2001. Larger problems can be solved rather easily with heuristic algorithms. These algorithms do not guarantee the optimality of the solutions but usually the results are near optimal (Knuutila *et al.* 2001).

The algorithms we present in this section rely on the same principles as those in see Knuutila *et al.* 2001, Knuutila *et al.* 2004 and Hirvikorpi *et al.* 2004. Inspired by the general formulation of section 3, the new algorithms are independent on the problem variant and their structure has been simplified. The use of multiple start points for local search in feasibility checks is also new and the *ITER-G* algorithm is original to this work.

4.1 Mapping of components

Depending on the problem variant, the mapping of component types to the feeder can be a simple test or an NP-hard problem. For the basic JGP all the component types are mapped to the only possible place, namely the linear feeder. In JGP-T components are mapped according to the component feeder types, which is still a trivial problem. JGP-B, however, complicates things essentially because in addition to mapping of the component types to the boxes one must choose a feasible set of boxes to the feeder organization. The algorithms presented later in this section use a heuristic (*createMapping*) to perform this task. Because these algorithms are suitable for all problem variants, this selection heuristic must be designed to handle also the most complicated problem JGP-B.

The problem of checking the feasibility of a group in JGP-B is a generalization of the bin packing problem. Hirvikorpi *et al*. 2004 use a two phase greedy heuristic (*fillBoxes*) to perform this test. In the first phase the simple boxes are filled, because they are able to accommodate components of certain width only. In the second phase the flexible boxes are filled in a greedy manner. This feasibility check is loose in the sense that the filling of the boxes is usually not optimal. However, the bin packing problem is NP-hard and the number of the feasibility checks made by the grouping algorithms is so large that a fast heuristic is required.

In addition to feasibility checking one must find a feasible feeder configuration for each group. Hirvikorpi *et al.* 2004 consider two different techniques to search for a suitable feeder organization. The other one is based on the brute force search, which simply tries all feasible box combinations. The second heuristic is iterative and it is based on local search (implemented in *improveConfiguration*). Both algorithms have two phases. In the first phase they choose some set of boxes and the second phase uses the greedy feasibility check to create the mapping. If the configuration does not pass the greedy feasibility check then of the algorithms move to the next box combination and repeats the process.

The combined mapping and feeder configuration search routine (*createMapping*) receives as an input the component type set, feeder parameters and one or several initial component type mappings, see figure 4. These initial mappings serve as a starting point for the local search (*improveConfiguration*).

```
createMapping(E, B, F, $f_1$, $f_{2,\ldots,}$ $f_n$) : configuration
    for i:=1 to n do                                 -- testing if any of the initial feeder configurations
        if |fillBoxes(E,$f_i$)|≤0 then return $f_i$;   -- is legal with respect to the given element set
    end for

    for i:=1 to n do
        f:=improveConfiguration(E,B,F,$f_i$);          -- doing local search for all initial feeder configurations
        if |fillBoxes(E,f)|≤0 then return f;          -- and testing their feasibility
    end for
    return ϕ;                                        -- no suitable feeder configuration was found
```

Figure 4. Construction of a feasible feeder box configuration. The routine returns a subset of boxes (feeder configuration) from *B* to which the element set *E* can be mapped legally. The feeder configurations $f_1, f_2, \ldots, f_n$ are used as starting points for the heuristic search of feasible feeder configuration. The heuristic search is implemented in *improveConfiguration* (Hirvikorpi *et al.* 2004). Fillboxes returns the set of component types left over from the configuration.

The generalized JGP heuristic allows temporarily violating the box capacity constraints and then repairs them (Knuutila *et al.* 2001). The concept of the feasibility of groups is therefore changed to a quantitative measure giving the degree of violation of the capacity constraints of a group or a single group. This measure is implemented in function *feasibility* which returns the sum of widths of the components left over from the given feeder configuration(s), see figure 5.

```
feasibility(G, B, F) : integer                      -- returns the feasibility of a grouping G with respect to
    sum:=0;                                         -- to the box collection B and feeder F
    for all groups $(g,f) \in G$ do                 -- sums the feasibilities of the groups within the
        sum:=sum+feasibility(E(g),F,B,f);           -- given grouping
    end for
    return sum;

feasibility(E, B, F, $f_1, f_2, \ldots, f_n$) : integer   -- calculates the feasibility of the given element set E
    min:=$\phi$;                                    -- with respect to the given box set B, feeder F
    for i:=1 to n do                                -- and the given initial feeder configurations $f_1, f_2, \ldots, f_n$
        f:=improveConfiguration(E,B,F,$f_i$);
        curr:=$\Sigma_{e \in fillBoxes(E,f)}$ w(e)
        if min=$\phi$ or else min>curr then min:=curr;
    end for
    return min;
```

Figure 5. Feasibility function for groups and groupings. Fillboxes returns the set of component types left over from the configuration.

4.2 Grouping by merging

The method in many efficient heuristics is first to group similar PCBs greedily and then use swap and move operations to improve the solution. The algorithm of figure 6 uses this method and it is applicable to all the three problem variants if the feasibility check is implemented according to the JGP-B and the simpler variants are transformed to this form the way described in section 3. The function *similarity*(g_1, g_2) returns the number of component types common to the given groups, *e.g.* *similarity*(g_1, g_2)=$|E(g_1) \cup E(g_2)|$.

```
GROUPER-G(C, B,F) : grouping                              -- creates a feasible grouping of C with
        $G:=\{\{(c, createMapping(c,B,F, \phi))\} \mid c \in C\}$;       -- respect to box set B and feeder F
        do
            $G_{new}:= \phi$;
            for all group pairs $(g_1,f_1), (g_2,f_2) \in G$ in order of descending similarity$(g_1,g_2)$ do
                $f:=createMapping(E(g_1) \cup E(g_2), B, F, f_1, f_2)$;
                if $f \neq \phi$ then
                    $g_m:=\{g_1 \cup g_2\}$;
                    $G_{new}:=(G \setminus \{(g_1,f_1),(g_2,f_2)\}) \cup \{(g_m,f)\}$;
                    break
                end if
            end for
            If $G_{new} \neq \phi$ then $G:=G_{new}$ ;
        while $G_{new} \neq \phi$
        return G
```

Figure 6. The pseudocode of *GROUPER-G*-algorithm.

The *GROUPER-G* algorithm performs the merge only if the resulting group is feasible. The solution can therefore be improved further by allowing the capacity constraints to be violated momentarily. The *ITER-G* algorithm performs merges which cause capacity restriction violations. It then tries swaps and moves of PCBs between the groups to make the grouping feasible again. The group pair to be merged is chosen so that the capacity overrun is minimal, see figure 7.

```
ITER-G(C,B,F) : grouping                                           -- a grouping algorithm which allows
                                                                   -- temporary capacity violations
    G:=GROUPER-G(C,B,F);                                           -- create an initial grouping
    Do
        merged:=false;
        for all group pairs $(g_1,f_1),(g_2,f_2)\in G$ in order of      -- try merging groups in order of
            descending feasibility($E(g_1\cup g_2)$, B, F, $f_1$, $f_2$) do     -- descreasing similarity
              $g_m:=\{g_1 \cup g_2\}$;
              $G_{new}:=(G \setminus \{(g_1,f_1), (g_2,f_2)\}) \cup \{(g_m,f_1)\}$;     -- merge groups and try to
              improveGrouping($G_{new}$, B, F);                          -- find a configuration for it
              if feasible($G_{new}$, B, F) then                          -- if configuration was found then
                  $G:=G_{new}$;                                          -- the merging was successful
                  merged:=true;
                  break
              end if
        end for
    while merged

improveGrouping(G, B, F) : grouping
    do
        improvement:=feasibility(G, B, F);
        for all $(g,f)\in G$, not feasible(g, B, F, f) do                -- try fixing the groups which are
            for all $c\in g$ do                                          -- not feasible
                for all $g_2\in G$, feasible($g_2$, B, F ,$f_2$) do
                    if feasible($\{c\}\cup g_2$, B, F, $f_2$) then
                        $g_2:=g_2 \cup \{c\}$;
                        $f_2$:=createMapping($E(g_2)$, $f_2$, B, F);
                        $g:=g \setminus \{c\}$;
                    end if
                end for
            end for
        end for
        improvement:=feasibility(G, B, F)-improvement;                   -- continue iterating as long as the
    while improvement>0;                                                 -- capacity overrun continues to
                                                                         -- decrease
```

Figure 7. A generic job grouping algorithm *ITER-G* which allows temporary capacity violations.

5. Empirical testing

One of the goals in the empirical tests is to find out how large problems can be solved exactly by the MILP-formulation of section 3. Another goal is to assess the efficiency of the problem variant independent algorithm of section 4. We use for all variants the same implementation of *ITER-G* and the problems are transformed to JGP-G according to the guidelines of section 3. The algorithm *ITER-G* is benchmarked against the best specialized algorithms from Knuutila *et al.* 2001, Knuutila *et al.* 2004 and Hirvikorpi *et al.* 2004. The third goal is to measure the increase in running time when moving from the basic JGP to JGP-B. All test runs were performed on a 2,0 GHz Pentium 4.

5.1 Test data

The test data were generated through randomization. First, we create a set of component types when the size of the set is given as a parameter. For each component type the width and the feeder type are selected from a given interval randomly. Next, the PCBs are created so that the number of component types on each PCB is selected randomly from the given interval. The component types are also selected random to the PCB.

Feeder generation for the basic JGP is rather trivial since the capacity is the only parameter. The capacity is calculated based on the PCBs created earlier and it is set to be twice as large as the largest PCB. The "size" of a PCB is the sum of the widths of the component types on it. For the JGP-T the feeder configuration is created in the same way as for the basic JGP except that the capacities of the feeders are calculated for each feeder type separately.

Box sets of JGP-B instances are created by selecting a combination of widths to each box. For example, assume that component types are of three different widths. Then, we create one box for each different width, one box for each different combination of two widths and finally one box which is able to hold the all widths. The capacity of each box is calculated as follows:

```
1. For all component widths $k \in W$
        Let $max(k)$ be $max\{w(c,k) \mid c \in C\}$.
2. For each box $b$
        Let $C(b)$ be the average of $max(k)$ where $k \in \mathrm{a}(b)$.
```

Component types in these boxes use more slots than their nominal width. The extra space used depends on the number of widths the box is able to hold. For two widths the components use 10 % more space than their nominal width and for three widths it is 20 % etc.

5.2 Exact solutions

Two common factors to different problem variants are the number of PCBs and the number of component types. The JGP-T adds the number of feeder types to these factors. In JGP-B the total number of boxes is the third essential factor. We measure the size of a problem according to the number of PCBs. For the JGP-T problem instances the number of feeder types is fixed to 3. For the JGP-B instances the number of boxes is either 3 or 6 and the feeder is able to hold 2 or 3 of these simultaneously, respectively.

The results of the first test run are summarized in table 1. It is observed that ILOG Solver was not able to solve problems of 20 PCBs and over in time limit of one hour without imposing an additional constraint. The following constraint was set for the JGP and JGP-T instances:

$$y_{kl} = 1 \qquad \text{for all } k=1,\ldots,|C| \text{ and } l=1,\ldots,U. \qquad (8)$$

Constraint (8) fixes the feeder configuration. This limits the size of the search quite drastically and pushes the limits of exact solving up 20 PCBs for JGP and 25 PCBs for JGP-T. Constraint (8) can not be set to the JGP-B instances because the feeder configuration is not fixed in them. For the JGP-B instances the search space had to be limited with tighter constraints. The following capacity constrain was also added:

$$\sum_{i=1,\ldots,|E|} e_{ik} m_{ikl} \le d_k y_{kl} \qquad \text{for all } l=1,\ldots,U,\ k=1,\ldots,|B|. \qquad (9)$$

This essentially says that the capacity of a box is 0 for certain group if it is not assigned to the box configuration of the groups. Although this constraint follows logically from constraints (6a) and (6b) this kind of redundancy is in some cases helpful.

Problem	**Number of PCBs**				
type	**10**	**15**	**20**	**25**	**30**
JGP	10	10	7	0 *)	0 *)
JGP-T	10	10	10	4 *)	0 *)
JGP-B	0**	0	0	0	0

*) additional constraint (8) was added to the constraint set to solve these

**) additional constraints (8) and (9) helped to solve problems of size 8 PCBs.

Table 1. The number of problems solved (out of 10) for different numbers of PCBs using the JGP-G model when implemented on ILOG Solver. The number of components is 5 times the number of PCBs, the number of feeder types is 3 and the number of boxes is 6. Timelimit is set to one hour.

5.3 Efficiency of the variant independent algorithm

The first test run for the *ITER-G* compares the solutions against best known problem variant specific algorithms *HG3* (Knuutila *et al.* 2001), *TS* (Knuutila et al. 2004) and *RedistrG* (Hirvikorpi *et al.* 2004), see table 2. The performance of the *ITER-G* is impressive since it is able to find better solutions for the JGP and JGP-T instances for all problem variants. The difference between *ITER-G* and *HG3* is statistically very significant.

Algorithm	**Problem variant**		
	JGP	**JGP-T**	**JGP-B**
ITER-G	13,00	9,70	12,77
HG3 (Knuutila *et al.* 2001)	14,20	-	-
TS (Knuutila *et al.* 2004)	-	9,77	-
RedistrG (Hirvikorpi *et al.* 2004)	-	-	12,57

Table 2. Comparison of *ITER-G* and the best known algorithms for different variants of JGP. The numbers in the cells are averages of the grouping sizes over 30 problem instances. The number of PCBs is 60 in each problem instance, there are 300 component types, 3 feeder types and 6 boxes. Timelimit of *HG3* is set to 2 minutes, tabu tenure for *TS* is 30 moves and time limit for *TS* is also 2 minutes. The difference between *HG3* and *ITER-G* is statistically very significant (paired *t*-test).

5.4 Comparison of running times

The running time of *ITER-G* depends strongly on the number of PCBs, the number of component types and the number of boxes. Table 3 shows the running time of ITER-G as a function of the number of PCBs and the number of boxes. One can see that the number of boxes and the number of PCBs scales the running time quadratically. Based on the results given by Hirvikorpi *et al.* 2004, the incresase in the running time would be exponential if this test would use the brute-force method to find the feeder configurations.

Number of boxes	**Number of PCBs**			
	30	**60**	**90**	**120**
8	0,12	1,55	6,28	18,22
9	0,13	1,66	6,09	14,43
10	0,19	1,41	5,38	15,71
11	0,25	1,74	6,40	17,28
12	0,37	2,20	7,26	18,63
13	0,55	2,88	8,56	19,68
14	0,83	3,87	10,35	22,19

Table 3. Running time of ITER-G in seconds as a function of the number of boxes and number of PCBs.

6. Conclusions

Three important variants of the job grouping problem were discussed. The variants differ with respect to the component feeder types and feeder unit organizations. Although the differences are apparently small they have a significant impact on the solution algorithms and the computation time.

We discussed the mathematical definitions of the variants. A mathematical program (JGP-G) was formulated. The program is able to solve all the variants when they are transformed suitably.The variant independent heuristic (ITER-G) is related to the variant dependent heuristic algorithms developed in earlier studies. A new idea in the search of feeder configurations is to use multiple starting points for the local search.

The empirical results show that the general mathematical program is able to solve equal sized problems as the previous variant specific programs. The heuristics were also shown to be statistically as efficient as the problem specific algorithms. The results also showed that the new heuristic is fast enough for practical sized problems.

The standard JGP is commonly somewhat more complicated than described here. Johnsson 1999 note that some of the fast chip shooters organize the linear feeder to several smaller so-called feeder banks. For example, certain component placement machines organize their 160 slot feeder to four feeder banks of 40 slots each. A complicating factor here is that the boundaries of the feeder banks can not be crossed by component packages. Depending on the component widths this may waste some feeder slots. Modelling this detail is superseded in previous studies by using smaller capacity for the feeder. It is interesting to see that JGP-B gives an efficient new way to formulate this open problem which has until now been largely bypassed.

Further research is needed to developed better lower bounds for the variants of the JGP. Another interesting topic of research is the definition of the problem size reductions; these have only been studied for the basic JGP.

Literature

Al-Fawzan M. A. and Al-Sultan K. S., 2003, A tabu search based algorithm for minimizing the number of tool switches on a flexible machine. *Computer & Industrial Engineering*, **44**(1), 35-47.

Bard. J. F., 1988, A heuristic for minimizing the number of tool switches on a flexible machine. *IIE Transactions*, **20**(4), 382-391.

Carmon T. F., Maimon O. Z. and Dar-El E. M., 1989, Group set-up for printed circuit board assembly. *International Journal of Production Research*, **27**(10), 1795-1810.

Crama Y. ja Klundert J., 1996, The approximability of tool management problems. Technical Report (rm96034), Maastricht Economic Research School on Technology and Organisation .

Crama Y., Kolen A. W. J. and Oerlemans A. G., 1994, Minimizing the number of tool switches on a flexible machine. *The International Journal of Flexible Manufacturing Systems*, **6**, 33-54.

Crama Y. and Oerlemans A., 1994, A Column Generation Approach to Job Grouping for Flexible Manufacturing Systems. *European Journal of Operation Research*, **78**, 58-80.

Hashiba S. and Chang T. C., 1992, Heuristic and simulated annealing approaches to PCB assembly setup reduction. In G. J. Olling and F. Kimura, editors. *Human Aspects in Computer Manufacturing*, IFIP Transactions B-3, 769-777.

Hertz A., Laporte G., Mittaz M. and Stecke K. E., 1998, Heuristics for minimizing tool switches when scheduling part types on a flexible machine. *IIE Transactions*, **30**(8),689-694.

Hirvikorpi M., Knuutila T., Johnsson M., Nevalainen O. S., 2004. Grouping of PCB Assembly Jobs in the Case of Flexible Feeder Units. To appear in *Engineering Optimization*.

Van Hop N., 2003, Board sequencing and component loading problem for a single machine in PCB assembly planning. *International Journal of Production Research*, **41**(18), 4299-4315.

Johnsson M., 1999, Operational and Tactical Level Optimization in Printed Circuit Board Assembly. Ph.D. Thesis, University of Turku, TUCS Dissertation 16.

Knuutila T., Puranen M., Johnsson M. and Nevalainen O. S., 2001, Three Perspectives for Solving the Job Grouping Problem. *International Journal of Production Research*, **39**(18), 4261-4280.

Knuutila T., Hirvikorpi M., Johnsson M. and Nevalainen O. S., 2004, Grouping PCB Assembly Jobs with Typed Component Feeder Units. To appear in *International Journal of Flexible Manufacturing*.

Kusiak A., 1991, *Handbook of flexible manufacturing systems* (Academic Press), pp. 147-194.

Leon V. J. and Peters B. A., 1998, A Comparison of setup strategies for printed circuit boards, *Computers & Industrial Engineering*, **34**(1), 219-234.

Maimon O. and Shtub A., 1991, Grouping methods for printed circuit boards. *International Journal of Production Research*, **29**(7), 1370-1390.

Shirazi R. and Frizelle G. D. M., 2001, Minimizing the number of tool switches on a flexible machine: an empirical study. *International Journal of Production Research*, **39**(15), 3547-3560.

Publication V

Hirvikorpi M., Knuutila T. and Nevalainen O., Job ordering and management of wearing tools in flexible manufacturing. To appear in *Engineering Optimization*, 2004.

Job Ordering and Management of Wearing Tools

M. Hirvikorpi, O. S. Nevalainen and T. Knuutila[*]

Department of Information Technology, University of Turku and Turku Centre for Computer Science (TUCS), Lemminkäisenkatu 14 A 20520 Turku, Finland

[*]corresponding author
timo.knuutila@utu.fi
phone: +358-2-3338635
fax: +358-2-3338600

Abstract

This paper considers a scheduling problem arising in the flexible manufacturing systems. It is assumed that a CNC machine processes a set of jobs with a set of wearing tools. The tool magazine of the machine has a given capacity and each job requires some subset of tools. The goal is to minimize the average completion time of the jobs by choosing their processing order and the tool management decisions intelligently. Previous studies concerning this problem have either omitted the tool wearing or assumed one tool type, only.

This study gives a mathematical formulation for the problem when the tool life-times are deterministic. We show that problems of practical size cannot be solved to optimality within reasonable time. We therefore consider genetic algorithms and local search methods to the problem. When the solutions of these new algorithms are compared against the optimal solutions and lower bounds, they are nearly optimal.

Keywords: CNC production, combinatorial optimization, job scheduling, tool wear, tool switches, heuristics

This work has been partially financed by the
Graduate School, Turku Centre for Computer Science (TUCS)

1. Introduction

The current trend in manufacturing is the widespread use of flexible manufacturing systems (FMS) which are able to manufacture variety of products with high quality and speed. In production planning one attempts to organize the production to as efficient as possible. The optimization of production efficiency is especially important in the fields of industry where competition is hard and therefore production planning has become an important area of research. In optimizing the production efficiency it is often useful to identify the bottlenecks of the process and to optimize the production from their perspective. The bottleneck in FMSes is in many cases a CNC machine which is used inefficiently.

In the *scheduling with wearing tools* (SWT) -problem we are given a set of jobs to be processed with a single CNC machine. The task is then to create a production plan which minimizes the average completion time of the jobs. The CNC machine uses limited life-time tools to process the jobs. The capacity of the tool magazine of the CNC machine limits the number of tools it can use simultaneously. Tools are removed (added) from (to) the magazine because their life-time has been consumed or because the job to be processed requires a tool which is not currently in the magazine. The problem is further complicated by the fact that one can switch tools between jobs, only. The reason for this is the possible degrade in quality and long fixed time associated to each interruption.

Scheduling with wearing tools was first considered by Akturk *et al.* (2002). Their problem formulation considered a case where there is only one tool type. We expand this model to include several tool types and a tool magazine. The motivation for studying this problem comes from the fact that tool switches due to the tool wearing are in some cases much more likely (ten times) than due to the part mix, see Gray *et al.* (1993).

Most of the studies concerning tool management and scheduling have concentrated on tool switches occurring due to the part mix. One exception here is the work of Akturk and Avci (1996) who developed a method for handling tool allocation and machining conditions. In their paper, the scheduling of CNC machines was not considered. It seems that the scheduling literature has omitted the tool management. On contrast to that, similarities to our study and the work by Akturk *et al.* (2002), can be found from the models which handle machine breakdowns and maintenance. Many of these models assume only one breakdown which in our case is not realistic since tool wearing occurs quite frequently. One exception to this is the study of Qi *et al.* (1999) who considers scheduling with multiple maintenance intervals which are of variable length. The model in Akturk *et al.* (2002) is equivalent to the model of Qi *et al.* (1999).

A traditional approach to tool management and scheduling has been to omit the completion times of the jobs from the minimization objective and to concentrate on the minimization of the number of tool switches. Tang and Denardo (1988) have developed 'Keep The Tool Needed Soonest' –policy which minimizes the number of tool switches for a fixed order of jobs. This problem is known as the tool loading -problem. The joint problem of job ordering and tool loading, called tool switching –problem, calls for a similar production plan as the SWT-problem but the objective is to minimize the number of tool switches. The tool switching –problem has been studied for example by Bard (1988), Crama *et al.* (1994) and Hertz *et al.* (1998). This traditional approach assumes that the job processing times are dominated by the tool switching times. Technological advances of the CNC machines have, however, reduced the switching times considerably thus giving additional movitation for studying the SWT-problem.

In some areas of production, like printed circuit board (PCB) assembly, a common approach has been to minimize the number of setup events. The job grouping problem (JGP) calls for a feasible minimal cardinality grouping of a given job set. A grouping is feasible if each group of jobs within the grouping can be processed without interruptions. The JGP setting assumes that setup events require some fixed time, i.e. the number of tool switches made during a setup does not affect to the duration of the event. To mention a few, the basic job grouping problem has been studied by Crama and Klundert (1996), Knuutila *et al.* (2001) and Smed (2002). Recent advances in the PCB assembly machines have made the basic JGP somewhat impractical. Knuutila *et al.* (2003) and Hirvikorpi *et al.* (2004) study new variants of the JGP. The operating principles of the modern component placement machines are considered more accurately in these variants.

In certain types of CNC production different tools require variable amount of space from the tool magazine of the CNC machine, a fact which many of the existing studies concerning tool management problems fail to model. In PCB assembly, for example, the components which are placed to the bare PCBs come in many different widths and it is not unsual that one component is three times as wide as some other component. Some of the recent studies concerning tool management problems take this fact to account, for example Tzur and Altman (2004) study the tool switching problem for tools of variable width. They also model the physical placement of the tools to magazine, see Chen *et al.* (2002) for the definition of this so called dynamic storage allocation problem. Matzliach and Tzur (2002) study the tool loading -problem for variable widths of tools but they do not model the physical placement of the tools to the magazine. As a continuation to their paper Hirvikorpi *et al.* (2004b) study the tool loading problem and dynamic storage allocation problem integrated, i.e. they also model the physical placement of the tools. In PCB assembly, Knuutila *et al.* (2001) and Hirvikorpi *et al.* (2004) model the variable width tools but not their physical placement.

Applications of the scheduling with wearing tools -problem can be found in PCB assembly and CNC machine production in general. In the context of PCB assembly the “jobs” are batches of PCB boards of the same type and the “tools” are electronic components which are placed on the board by a component assembly machine. The life-time of a tool is in this case the number of components of same type which can be loaded into the feeder unit (corresponding to the magazine). In CNC production, the tools can be for example drill bits or cutting tools and jobs are different types of industrial parts which are processed by a CNC machine.

It is well known that the Shortest Processing Time (SPT) –rule (Brassard and Bratley 1996) minimizes the average completion time for a single machine if we omit tool wearing and tool switches. In this paper we formulate the SWT-problem and show that the SPT-rule does not perform well if tool wearing and tool switches are taken into account. Because of this, we introduce more elaborate heuristic methods for solving the SWT –problem and show that they perform very well with respect to optimal solutions. We also consider lower bounding –methods and give an optimal solution to our problem.

The rest of the paper is organized as follows. Section 2 defines the s*cheduling with wearing tools* -problem (SWT) formally. Section 3 presents an integer programming formulation for the problem. Section 4 introduces two heuristic algorithms based on evolutionary and local search methods. Section 5 discusses different lower bounding methods for the SWT-problem. Empirical results about the efficiency of the heuristic methods are presented in Section 6. The paper is closed with some concluding remarks in Section 7.

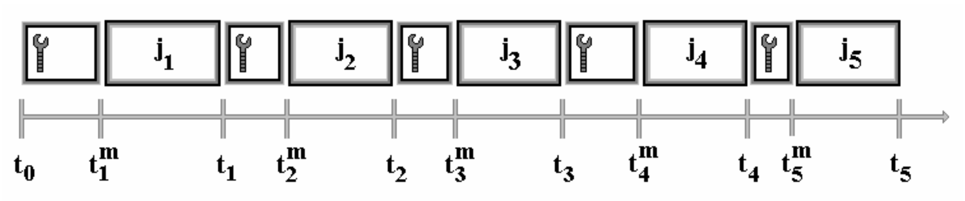
2. Problem definition

A central part of the problem definition in the field of production planning and control is the machine abstraction. Because CNC machines vary greatly from the technological perspective, our abstraction fixes a minimal set of assumptions about the working principle of the machine. First, we assume that the CNC machine uses wearable tools to process jobs and that the machine is able to use only the tools which are in the tool magazine. Capacity of the tool magazine indicates how many tools it can hold simultaneously which means that each tool consumes a uniform amount of space from the tool magazine. Furthermore, tools which are not in the magazine are stored in the vicinity of the machine and a time cost (called switching cost) is associated with each tool addition and removal. Finally, the structure of the magazine is assumed to be such that the location of a tool in the magazine does not affect to the switching time.

We say that a tool is a *physical instance* of some *tool type* and associate with each tool type a deterministic life-time and a switching cost. Life-time is the time of proper functioning for a tool of certain tool type and switching cost indicates the time required to add (remove) this type of tool to (from) the magazine. Life-time depends only on the tool type, an assumption which eliminates the variance on the time of proper functioning between different physical instances of a tool type. These life-times can be approximated for CNC machines when making certain assumptions on the conditions of the job processing, see Akturk and Avci (1996).

Each job requires some set of tool types in processing and therefore we define for each (job, tool type)–pair a deterministic processing time. We assume that this processing time is less than the life time of a tool type for all pairs, otherwise the job in question could never be processed. The time required to process a job is the sum of the individual (job, tool type) –pair processing times. Further, once the processing of a job has began, it must be completed without interruptions, i.e. no tool switches to the magazine of the machine are allowed. Technically the switches may be possible but each interruption of the machine requires in addition to individual tool switching times a certain fixed time which can be an order of magnitude longer than the tool switching times. In some cases there is also a possible degrade in quality when the processing is interrupted, therefore these interruptions are not allowed.

Let *linear time* be the sum of processing times of all jobs in *J* and *quadratic time* the sum of completion times of the jobs in *J*, see Fig. I. The *average completion time of the jobs* in *J* is the quadratic time divided by the number of jobs (|*J*|). The objective in the SWT-problem is to minimize this time.



$$c_l = t_5$$

$$c_q = \sum_{i=1}^{5}(5-i+1)t_i + \sum_{i=1}^{5}(5-i+1)(t_i^m - t_{i-1})$$

Figure I. The concepts of *linear time* c_l and *quadratic time* c_q when processing the jobs $J=\{j_1, \ldots, j_5\}$. The schedule includes some tool operations like tool addition and removal between the jobs.

We use the following notations to define the SWT-problem formally:

T set of tool types,

c_i time required for tool addition (removal) to (from) the magazine for tool type $i \in T$,

	l_i	life-time of tool type $i \in T$,
J	set of jobs,	
	p_j	total processing time of job $j \in J$,
	$p_{i,j}$	processing time of job $j \in J$ for tool type $i \in T$,
	T_j	the set of tool types required by job $j \in J$, $T_j \subseteq T$,
K	capacity of the tool magazine.	

Magazine state $S=(M,\alpha)$ is an ordered pair where M is the set of tool types currently in the magazine and function $\alpha{:}T \rightarrow N$ where $i \in M$ gives the remaining life-time of the tool of tool type i. If $i \notin M$ then $\alpha(i)=0$. An implicit consequence of this definition is that there is at most one physical instance of certain tool type simultaneously in the magazine. S_0 is used from now on to indicate an empty magazine state.

State switch cost $c(S_1,S_2)$ for the two magazine states $S_1=(M_1, \alpha_1)$ and $S_2=(M_2, \alpha_2)$ is the total time used for tool removals, additions and replacements (changing worn out tools) to switch state S_1 to state S_2 and it is defined as follows:

$$c(S_1,S_2) = \sum_{i \in M_1 \oplus M_2} c_i + 2\sum_{i \in M_1 \cap M_2 \text{ and } \alpha_1(i) < \alpha_2(i)} c_i \,.$$

The first term of $c(S_1, S_2)$ sums up the costs of the removed and added tools and the second one counts for the costs of the replaced tools. Notice that tool replacement is considered as a two phase operation, since the tool must first be removed and then added. State switches are made between the jobs.

Job j is *feasible* with respect to magazine state $S=(M, \alpha)$ if and only if $\alpha(i) \geq p_{i,j}$ for all $i\in T_j$, i.e. the tools of the magazine state S have enough life-time left to process the job j.

A magazine state is *legal* with respect to capacity K if and only if $|M|\leq K$. It is assumed here that each tool uses the same amount of capacity and therefore the capacity is calculated simply as the number of tools the magazine can hold simultaneously.

State transition is defined as a triplet (S_1,j,S_2), where S_1 and S_2 are magazine states and j is a job. State transition describes the change occurring in magazine state when processing job j.

We say that transition (S_1,j,S_2) where $S_1=(M_1, \alpha_1)$, $S_2=(M_2, \alpha_2)$ and $j\in J$ is *legal* if and only if:

1) j is feasible with respect to S_1,
2) $\alpha_1(i)= \alpha_2(i)+p_{i,j}$ for all $i\in T_j$,
3) $\alpha_1(i)= \alpha_2(i)$ for all $i\in M_1\setminus T_j$,
4) $M_1=M_2$.

Condition 1 guarantees that the initial magazine state S_1 is feasible with respect to job j, otherwise it cannot be processed with magazine state S_1. Conditions 2 and 3 enforce that the tools wear out properly during the processing of j. Finally, condition 4 guarantees that tools are not added (removed) to (from) the magazine during the processing of j.

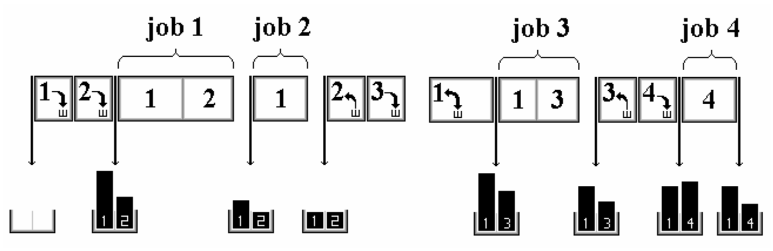


Figure II. An example of a production plan of 4 jobs, 4 tools and a magazine of capacity 2. The numbers inside the boxes indicate which tool type is removed, added or used in the processing of a job. For tool switches the direction of the arrow indicates the type of tool switch: arrow down is tool addition, arrow up is tool removal and two-headed arrow is tool replacement. The magazine state in each phase is described below the processing order and the length of the bar illustrates the remaining life-time of the tool.

Production plan P is an ordered set of state transitions $P=\{(S_1,j_1,S_2), \ldots, (S_{2n-1},j_n,S_{2n})\}$. We say that production plan P is *legal* with respect to capacity K and job set J if and only if:

1) $j_1, j_2, \ldots, j_n$ is a permutation of J,
2) state transitions $(S_1,j_1,S_2), \ldots, (S_{2n-1},j_n,S_{2n})$ are legal,
3) magazine states $S_1, \ldots, S_{2n}$ are legal with respect to capacity K.

An example of a production plan is given in Fig. II. This particular plan is for a problem instance of 4 jobs, 4 tools and capacity 2 tool magazine.

Cost function $C_{plan}(P)$ for production plan $P=\{(S_1,j_1,S_2), \ldots, (S_{2|T|-1},j_{|T|},S_{2|T|})\}$ is:

$$C_{plan}(P) = \sum\nolimits_{k=1,\ldots,|T|} (|T|-k)p_{j_k} + \sum\nolimits_{k=0,\ldots,|T|-1} (|T|-k)c(S_{2k,2k+1})$$

(1).

Formula (1) calculates the quadratic time (see Fig. 1) of the production plan *P*. Because the completion times of the jobs are not directly available in the production plan they must be summed up. The processing time of the first job p_{j_1} delays all the $|T|$-1 jobs after it by the time p_{j_1} so it must be multiplied by $|T|$-1. The tool switches before the first job delay all the jobs by the time $c(S_0,S_1)$ thus it must be multiplied by $|T|$. The contribution of each job to the quadratic time is summed up similarly in formula (1).

Scheduling with wearing tools problem (SWT): given tool type set *T*, job set *J* and capacity *K* find a production plan *P* which is legal with respect to capacity *K* and job set *J* and $C_{plan}(P)$ =min!

Formula (1) is quite similar to the objective function of the single tool problem studied by Akturk *et al.* (2002) but the cost calculation between the jobs is more complex due to the multiple tool types and variable tool type switching times.

We defined for each job *j* the total processing time p_j and for each tool *i* the processing time $p_{i,j}$. If we assume that $\sum_{i \in T_j} p_{i,j} = p_j$ then we do not need the value p_j. We should not however make this assumption in general because it is possible that the job is processed with several tools at the same time (c.f. PCB assembly with a dual-cantry placement machines). It is also possible that some jobs require longer preparation or post-processing than others. These times can be included in the job processing time.

If we consider the single tool problem presented in Akturk *et al.* (2002) we observe that it is a special case of the SWT-problem stated above. This can be seen as follows. Let us assume that we are given an instance of the single tool problem with

T_C	tool switching time,
p_j	processing time of job j where j=1,…,k.

Our task is then to sequence the n jobs and perform the tool switches in such a way that the average job completion time is minimized. This can be transformed to an instance of the SWT-problem simply by defining:

Tool set $T=\{1\}$, $c_1=T_C$ and $l_1=T_L$,

Set of jobs $J=\{1, 2,\dots,k\}$ and for all $j\in J$: $p_1 j=p_j$ and $T_j=\{1\}$,

Capacity of the magazine $K=1$.

Then, an optimal solution to SWT-problem instance would give an optimal solution to the original problem instance. It has been shown by Akturk *et al.* (2002) that the single-tool change problem is NP-hard and therefore the SWT-problem is also NP-hard.

3. MILP formulation for the SWT-problem

The MILP formulation for the SWT-problem is useful when evaluating the efficiency of the heuristic algorithms of Section 4. It will also also be used as a basis of the Lagrange relaxation described in Section 5. Let us first define the problem parameters:

T_{count}	number of tools (i=1,…,T_{count}),
J_{count}	number of jobs (j=1,…,J_{count}),
l_i	lifetime of tool type i tool,
c_i	time required to add or remove tool type i tool
p_j	total processing time of job j,

$p_{i,j}$	processing time of job *j* for tool type *i*,
$h_{i,j}$	1 if job *j* requires tool type *i*, 0 otherwise, and
K	number of tools the magazine can hold simultaneously.

The decision variables are (t=1,…,J_{count}):

$x_{j,t}$	1 if job *j* is processed at time *t*, 0 otherwise,
$b_{i,t}$	remaining life-time of the tool type *i* at time *t*, before processing t^{th} job,
$a_{i,t}$	remaining life-time of the tool type *i* at time *t*, after processing t^{th} job,
$z_{i,t}$	1 if tool type *i* is in the magazine at the time *t*, 0 otherwise,
$s_{i,t}$	1 if tool type *i* is removed or added at the time *t*, 0 otherwise,
$r_{i,t}$	1 if tool type *i* is replaced at time *t*, 0 otherwise.

The SWT-problem can then be stated as:

$$\sum_{t=1,\dots,J_{count}} \sum_{j=1,\dots,J_{count}} (P-t)x_{j,t}p_j + \sum_{t=1,\dots,J_{count}} \sum_{i=1,\dots,T_{count}} (P-t+1)(s_{i,t}+2r_{i,t})c_i = \min! \qquad (2)$$

s.t.

$$a_{i,t} = b_{i,t} - \sum_{j=1,\dots,J_{count}} x_{j,t}p_{i,j} \qquad i=1,\dots,T_{count},\ t=1,\dots,J_{count} \qquad (3)$$

$$\sum_{j=1,\dots,J_{count}} x_{j,t} = 1 \qquad t=1,\dots,J_{count} \qquad (4)$$

$$\sum_{t=1,\dots,J_{count}} x_{j,t} = 1 \qquad j=1,\dots,J_{count} \qquad (5)$$

$$z_{i,t} \geq \sum_{j=1,\dots,J_{count}} x_{j,t}h_{i,j} \qquad i=1,\dots,T_{count},\ t=1,\dots,J_{count} \qquad (6)$$

$$s_{i,t} \geq z_{i,t} - z_{i,t-1} \qquad i=1,\dots,T_{count},\ t=1,\dots,J_{count} \qquad (7)$$

$s_{i,t} \geq z_{i,t\text{-}1} - z_{i,t}$ $\qquad i=1,\ldots,T_{count},\ t=1,\ldots,J_{count}$ (8)

$s_{i,t} \leq (1\text{-}z_{i,t})+(1\text{-}z_{i,t\text{-}1})$ $\qquad i=1,\ldots,T_{count},\ t=1,\ldots,J_{count}$ (9)

$r_{i,t} \geq (b_{i,t} - a_{i,t\text{-}1}) / l_i - s_{i,t}$ $\qquad i=1,\ldots,T_{count},\ t=1,\ldots,J_{count}$ (10)

$\sum_{i=1,\ldots,T_{count}} z_{i,t} \leq K$ $\qquad t=1,\ldots,J_{count}$ (11)

$z_{i,t}\, l_i \geq a_{i,t},$ $\qquad i=1,\ldots,T_{count},\ t=1,\ldots,J_{count}$ (12a)

$z_{i,t}\, l_i \geq b_{i,t},$ $\qquad i=1,\ldots,T_{count},\ t=1,\ldots,J_{count}$ (12b)

$x_{j,t},\ z_{i,t},\ s_{i,t},\ r_{i,t} \in \{0,1\},$ $\qquad i=1,\ldots,T_{count},\ j,t=1,\ldots,J_{count}$ (13)

$a_{i,t},\ \geq 0,$ $\qquad i=1,\ldots,T_{count},\ t=1,\ldots,J_{count}$ (14a)

$b_{i,t} \geq 0,$ $\qquad i=1,\ldots,T_{count},\ \ t=1,\ldots,J_{count}$ (14b)

$z_{i,0} = a_{i,0} = 0$ $\qquad i=1,\ldots,T_{count}$ (15).

Objective function (2) calculates the quadratic time and is essentially same as function (1). Constraint (3) binds the transition variables so that tool wearing actually occurs. The next two constraints guarantee that every job is processed exactly once. Constraint (6) says that all the tools required during each time period must be in the magazine. The following constraints (7) and (8) enforce that whenever a tool is added or removed its decision variable is set to 1. Constraint (9) prevents the addition of a tool when a replacement of the tool type in question is needed. This must be enforced because tool addition is cheaper than tool replacement. Tool replacement occurs when tool removal or addition is not made, but the life-time of the tool increases from the previous time period (10). Constraint (11) says that the given magazine capacity must not be exceeded. If tool type is not in the magazine its life-time must be 0, this is formalized with constraint (12). The constraints (13) and (14) set the domains for the decision variables and constraint (15) says that the processing starts with an empty tool magazine.

Note that in this formulation a tool is always replaced by a new one when removed from the magazine. This is the case even if its remaining life-time were long enough to allow further use. A simple instance of the SWT-problem is shown in Fig. III.

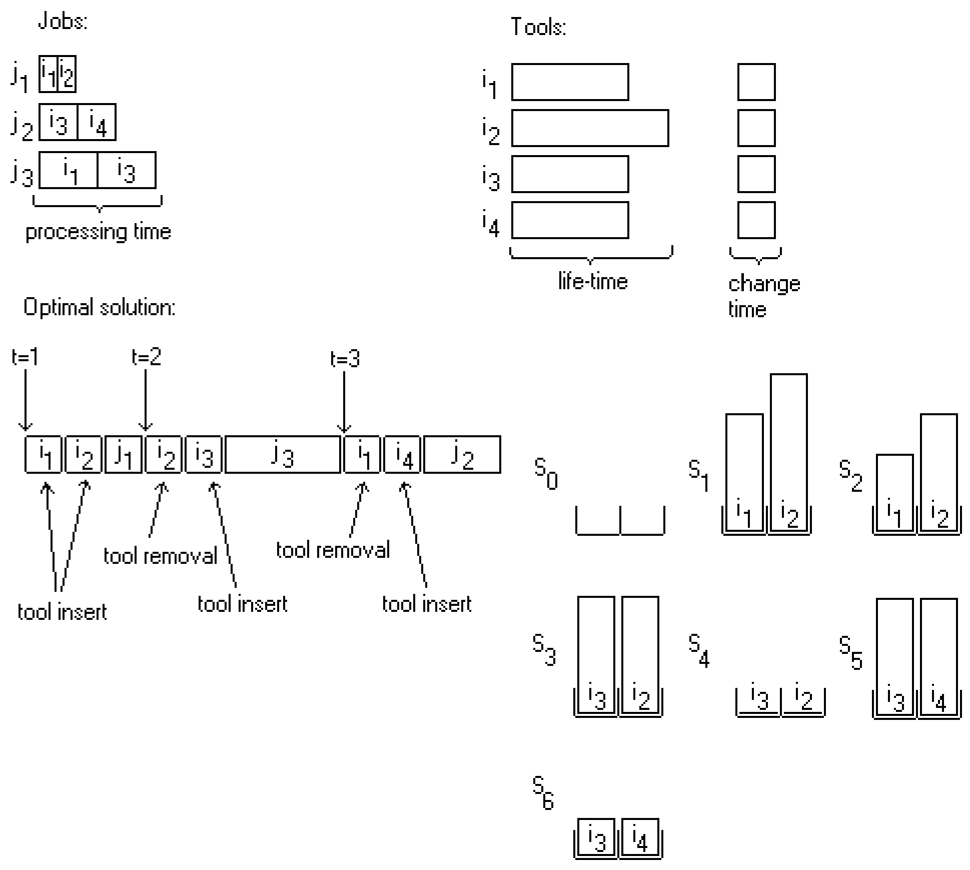


Figure III. Simple SWT-problem instance. The capacity of the tool magazine is K=2.

4. Heuristics

This section introduces two different methods for solving the SWT-problem. The first one is a simple local search which starts with the SPT-order and improves it by making changes to the job processing order. The second method is a genetic algorithm which utilizes randomness in the search. Both of these methods are guided by a cost function which calculates the average cost for a certain order of jobs. The cost function must find the optimal tool switches for a fixed job processing order in order to minimize the average job processing time. To solve this, what we call the *tool scheduling* –problem, the cost function applies a tool switching policy which we develop first.

4.1 Tool switching policy

The tool scheduling -problem is trivial if the magazine can accommodate only one tool at a time (and each job requires one tool only), see Akturk *et al.* (2002). In this case the tool is switched when it can no longer process the next job in the order. This happens when it's life-time expires or the next job requires a different tool. However, in the case of multiple tool types and a tool magazine capable of storing several tools simultaneously, we have more degrees of freedom and the costs depend on the decisions in the management of the magazine.

The solution methods for the SWT-problem we present later require that the cost function which solves the tool scheduling –problem is fast. This is due to the fact that one run of the genetic algorithm (GA) may make hundreds of thousands of cost evaluations. A greedy heuristic which proceeds iteratively through the job sequence and makes locally best tool switches is therefore our choice. There are three types of situations tool switching policy must solve:

1) tool which is needed by the next job is not in the magazine,
    a) there is space in the magazine for the tool or
    b) the magazine is full,
2) tool is worn out, i.e. it is not able to process the next job,
3) tool is able to process the next or is not needed by the next job.

The cases 1a, 2 and 3 are trivial. In case 1a our tool switching policy simply adds the new tool to the magazine and in case 2 the worn out tool is replaced with a new tool. In case 3 no action is necessary. The case 1b is more complex because we must remove a tool which is currently in the magazine. The Keep Tool Needed Soonest (KTNS) –rule is optimal for the basic tool loading –problem. It is easy to see that the KTNS-rule is not optimal in the case of wearing tools when minimizing the average processing time. For this, consider a case where there is a completely worn out tool in the magazine, i.e. this tool cannot process any more jobs. It is then more cost-efficient to remove this tool than any other tool with the same removal cost. KTNS would not necessarily select this tool because it does not consider the remaining life-time of tools at all. Another problem with KTNS is that it does not take the removal costs of the tools into account either. Based on this we formulate a revised removal rule.

The easiest way to incorporate the remaining life time of tools to the removal rule is to compare the wearing out times instead of the times when they are needed the next time. We propose a rule which Keeps the Tool which Wears out Last (KTWL). The wearing out times, which are calculated from the jobs still to be processed, are also divided by the switching costs. The tools which are not used any more are naturally put at the front of the list of removals. This kind of rule has several appealing properties:

- completely worn out tools are removed first according to their switching costs,
- tools with long life-times are removed rarely from the magazine,
- tools with the same usage distance are selected based on the removal cost,
- both of the new factors, life-time and switching costs, are taken into account.

A problem with this rule is that if the life-times are long in comparison to the processing times then all tools can have the same wearing-out time. The KTWL-rule chooses the tool with the lowest switching cost. This means that the rule makes an implicit assumption that the tools are not removed from the magazine before they are completely worn. The smaller the magazine is with respect to the number of tools and the longer the life-times, the worse this rule will work. The rule can be improved by combining the KTNS and KTWL rules to a HYBRID-rule by the formula:

$$d_{hybrid}(i,r,O,t)=(f^*d_{ktns}(i,O,t) + (1\text{-}f)^*d_{ktwl}(i,r,O,t)) / c_i$$

(16).

In this formula O is the processing order of the jobs, t is the current index in O, i is the tool type for which we are calculating the rule value and r is the remaining life-time of the tool of type i. Functions d_{ktns} and d_{ktwl} calculate the values of the KTNS and KTWL rules. The function d_{ktns} calculates the distance to the next job using tool type i and d_{ktwl} calculates the distance when the tool (i,r) will wear out. These functions quantify their values as number of jobs, for example if d_{ktwl} returns value 2 then the tool (i,r) wears out before the t+3th job. Parameter f is a weighting factor for the two rules.

We are now ready to formulate our heuristic for the tool scheduling –problem (*SWTC*, see Appendix A for implementation). Let us assume that the current magazine state is S and the next job of the sequence is j. For every tool type i in T_j there are three main cases which we outlined above. In case 1 the heuristic adds a tool of tool type i to the magazine if it isn't full. If the magazine is full, the removal rule is applied to select a tool which is then replaced. In case 2 the tool is replaced with a tool of same type and in case 3 no action is necessary. The SWTC-algorithm proceeds iteratively through the job sequence given by the parameter *jobs* and maintains the magazine state. The parameter K is the capacity of the magazine and R gives the rule to be used (KTNS, KTWL or the weighted combination of these – HYBRID). Upon return the value of SWTC is the quadratic time of the job sequence.

4.2 k-optimal local search

The SPT-rule (shortest processing time first) together with our tool switching policy solves the SWT-problem heuristically. This simple rule ignores the life-times and magazine allocations of the tools totally. Although the average cost for the SPT-order may be far from optimal, this order can still be used as a starting point for local search algorithms.

The k-optimal local search algorithm (LS(k)) searches in the neighborhood of the currently best solution and greedily proceeds to the direction of better solutions according to the SWTC-algorithm. The neighborhood consists of all orderings which can be reached with a chain of the length at most k changes. A change is either a swap of two jobs in the sequence or a move of a job into a new position. The maximal chain length k is given as a parameter. LS(k) proceeds greedily to the best possible new solution in the neighborhood of the current solution if it is better than the current one. The same is reiterated until no improvement is found.

4.3 Genetic algorithm

The GAPSM algorithm described in this section is a modification of the GAPS, which was developed by Akturk *et al.* (2002) for the single tool problem. Interested reader is referred to Reeves 1995 (pp.151-188) for a general description of a genetic algorithm (GA) and Akturk *et al.* (2002) for the description of the original GAPS algorithm. GAPSM has many technical details which are not of great importance here, therefore only an overall description of the algorithm is given. The key components of a GA include coding an individual, fitness evaluation, crossover, mutation and selection. These components are described next and after that a brief description of the main phases of GAPSM is given.

In the SWT-problem a set of jobs J is given as an input. Each individual of GAPSM population describes some permutation of the jobs in J. In order to make our GA work, the individuals must be coded so that they can describe all the $|J|!$ possible permutations. The coding of an individual is based on the idea of problem space search, see Storer *et al* (1983); an individual appears as a vector of perturbation factors, which are real numbers from the interval (0,1]. These are used as multipliers of the processing time of the jobs. Now, it is possible to use almost any kind of deterministic base heuristic to calculate an order for the jobs from these perturbed processing times. Determinism means here that when given certain processing times, the heuristic produces always the same order. GAPSM uses the SPT-rule which sorts the jobs in order of increasing processing times. It is easy to see that by choosing the perturbation factors properly, one can generate any of the permutations of the jobs by applying the SPT-rule to these perturbed processing times. In this way, an individual which is an ordering of the jobs is described fully by two components: a perturbation vector and the base heuristic.

The fitness of an individual is evaluated in two steps. First, GAPSM creates with the SPT-rule (our base heuristic) a job sequence. The processing times which have been multiplied with the perturbation vector of the individual are used in the first step. In the second step, the result of the first step serves as an input to the SWTC-algorithm. Its result (the average processing cost for the given order) is the fitness of the individual. This is the major difference between GAPS (Akturk *et al.* 2002) and GAPSM, since GAPS uses a different kind of fitness evaluation heuristic.

In the crossover, parents are chosen by tournament selection. The number of tournament rounds is selected randomly from an interval which is given as a parameter. For example, if the number of tournament rounds is 0 then the selection of the parents is purely random. After the selection of the parents, the algorithm takes the first half of the perturbation factors from the other parent and the second half from the other. These halves are then catenated to form an offspring.

Each individual in the population is allowed to mutate with a small probability during each iteration of the GAPSM. When a mutation event occurs, the algorithm selects one random position in the perturbation vector of the individual in question and replaces it with a random number from interval (0,1].

During each iteration of GAPSM one offspring is created in the manner describe above. This offspring then replaces the worst individual (i.e. the one with the highest average processing cost) of the current population. All the other individuals are directly transferred to the next iteration.

GAPSM algorithm begins by creating an initial population of certain size (a parameter). The perturbation factors of the initial population are chosen from the uniform distribution of (0,1]. In the evolution phase GAPSM forms a new generation of the individuals as described above. The evolution phase continues given number of iterations and the final result of GAPSM is the job sequence of the individual with the best fitness after the last iteration.

4.4 Using local search in GA

There are several ways to incorporate the *k*-optimal search to the GA of section 4.3. The most obvious and simple way to improve the population is to apply the *k*-optimal algorithm to the individuals of the final population. GAPSM2 uses this technique because there is no guarantee that the individuals of the population are even locally optimal when GAPSM finishes.

Instead of doing post-processing like in GAPSM2, one can try to do the improvements along the way as GAPSM iterates. This, however, imposes the question of how to change the individuals. One must notice that changing the perturbation factors of the individuals is the only way to modify the order of the jobs due to the use of the base heuristic (see section 4.3). This also means that the local search algorithm of section 4.2 can not be used directly. One cannot swap the factors directly in order to swap two jobs in the job sequence: proper rescaling of the factors is necessary.

The algorithm GAPSM3 subjects a given percentage (a parameter) of the population for improving moves at each iteration. These improving moves replace the mutations of GAPSM. In the manipulation, a single improving move for two sequential components of the perturbation vector is made. Move operation manipulates the perturbation factors of the jobs in a way which makes them swap their positions when the base heuristic is applied. More general swaps are avoided in order to cut down running times. The GAPSM3 uses also the *k*–optimal search after the evolution phase to find the (local) optimums for the individuals of the final population.

5. Lower bounds for the SWT-problem

The exact solution formulation of section 3 uses a plenty of 0/1-variables; $|J|^2+5|J||T|$ in total. Because of this and the fact that the SWT-problem is NP-hard, we find it highly unlikely that optimal solutions for problems of reasonable size (e.g. 100 jobs, 20 tools) are found within bearable timelimits.

A simple way to calculate the lower bounds is to use a linear relaxation of the problem. In linear relaxation one simply replaces all integer variables with real variables. After this the linear problem can be solved with the standard simplex algorithm which is normally efficient enough for practical sized problems. The problem with this approach is that the bounds found in this way are often quite loose.

5.1 Lagrange relaxation

Another more sophisticated and efficient method is the Lagrange relaxation (see Reeves 1995, chapter 6 for example). The idea is to relax some of the constraints and move them to the objective function. The relaxed constraints are multiplied with factors which affect to the quality of the lower bound. It can be shown (using the dual problem) that the optimal value for the relaxed problem is always a lower bound for the original problem. In the following, we use the subgradient method for tuning the constraint multipliers.

The decision variables *x* and *z* are the ones we are actually looking for in the model of section 3. The other variables can either be derived from these or are not of interest. We studied several different relaxations but none of them gave significant improvements in running times or, if running times were fast, the quality of the bound was not significantly better than with the linear relaxation. The first attempt was to relax the constraints (4) and (5). These were moved to the objective function (1) in the form:

$$u_t(1-\sum\nolimits_{j=1,\dots,J_{count}} x_{j,t}) + v_t(1-\sum\nolimits_{t=1,\dots,J_{count}} x_{j,t}) \tag{17}$$

where *u* and *v* are the Lagrange multipliers for these constraints. This relaxation gave significant improvement in the running times but the lower bounds were loose, only a bit tighter than using the linear relaxation. Several other relaxations were tested but none of them worked satisfactorily.

5.2 The SPT-order

The SPT-order is optimal if we omit the tool switches. This gives rather weak lower bound when the time of tool switches take a considerable time with respect to the processing time. In order to be able to add the tool switches to the schedule we need to simplify the problem so that the SPT-order is optimal even with the tool switches. The simple example in Fig. IV shows that even if the capacity of the magazine is considered to be infinite and the processing of a job can be interrupted, the SPT-rule is still suboptimal.

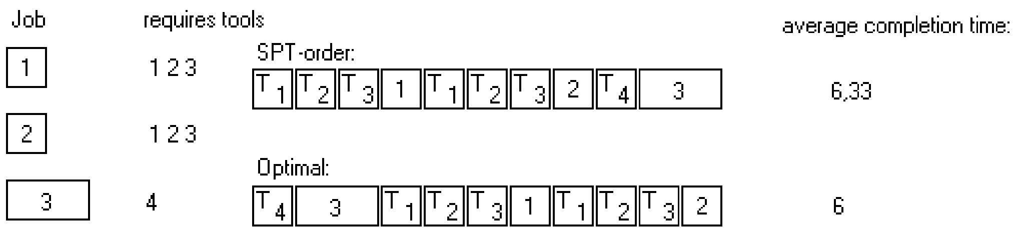


Figure IV. The suboptimality of the SPT-order in a simplified SWT-problem instance where the tool switching time is 1 for all tools. In the schedule $T_{1,\dots,4}$ is a tool add. Tool life-times are 1, 1, 1 and 2.

The suboptimality is caused here by the multi-tool property of the problem. In order to make the SPT optimal we would have to simplify the problem to a single tool problem and allow processing interruptions.

6. Empirical results

We wanted to find out how large problems can be solved to optimality within reasonable time by implementing the mathematical model of section 3. Another goal was to assess the heuristics of this research against the optimal solutions. This was, however, possible for small problem instances, only. We therefore compare the heuristics experimentally for problems of larger size.

The problem instances were generated with randomization. For the data generation process the following parameters were given:

- number of jobs $|J|$,
- number of tools $|T|$,
- capacity K,
- interval for the number of tools per job $Tools_{job}$,
- interval of processing times of jobs Job_{times},
- interval of tool life-times $Tool_{life\text{-}times}$,
- interval of tool switching times $Tool_{switch\text{-}times}$.

Uniform distribution was used for all the intervals in the data generation process. The program definition described in section 3 was implemented with the ILOG solver.

The purpose of the first test run was to evaluate how large problem instances can be solved to the optimality within reasonable time. This information is important for two reasons. First, we need it when designing the remaining test runs, especially when evaluating the heuristics againts the optimal solutions. Secondly, it is interesting to see whether optimal solution of problem instances of practical size can be expected. Two critical factors affect to the number of decision variables of the MILP formulation: tool set size and the number of jobs. We therefore consider three types of problem instances: high-, medium- and low tool usage problems. In high tool usage problems the number of different tools used by the jobs is small and in low tool usage problems the number different tools used by the jobs is large. In the high tool usage problems the size of tool set is 10% of the size of the job set. For medium and low usage problems these percentages are 20 and 40. Table I shows the percentage of the problems which the

ILOG system was able to solve successfully. All tests were performed on Pentium 4 2,0GHz, 512 Mb main memory and Windows 2000.

Table I. Exact solution of SWT-problem. The table shows the percentage of the problem instances (out of 30) that ILOG solver was able to solve to optimality within a limit of one hour. The parameters used in the generating process of this test data are summarized in appendix B.

Number of jobs				
Tool usage	**10**	**12**	**16**	**20**
High	100 %	100 %	100 %	3.3 %
Med	100 %	90 %	3.3 %	0.0 %
Low	97 %	43 %	0.0 %	0.0 %

In test run 2 the aim was to find out how the different tool removal rules affect to the processing costs. In the heuristic algorithms the choice of the rule guides the search and this is why we decided to solve the costs of 1000 random orders for each problem instead of searching for the best possible orders. This kind of test makes it possible to evaluate the tool switching policy as isolated from the search procedure. The hypothesis was that for smaller problems in which the tool life times are long in comparison to the total completion time, the KTNS works better than KTWL. This is important for later test runs, because we would like to find out the "best" tool switching policy. It would allow us to use the same tool switching policy for all runs. The hybrid rule based on the KTNS and KTWL was therefore tested with several factors and the best one is reported in Table III. From the results we can see that the KTNS is always worse than the hybrid or KTWL. Surprisingly, the KTWL is in most cases better than the hybrid rule, although it has some obvious flaws. The optimal setting for the factor of the hybrid rule in this test run is between 0.35-0.6. Inside this interval the differences were statistically insignificant.

Table II. Effect of the removal rules to the average cost. The test data was generated with the following parameters: $|J|=\{20,30,50,100\}$, $T=10$, $K=4$, $Tools_{job}$~U(1,2), $Tool_{life\text{-}times}$~U(7,9) and $Tool_{switch\text{-}times}$~U(2,3). The numbers are averages of quadratic times of 1000 random orderings.

	Number of jobs							
	20		**30**		**50**		**100**	
Rule	average	stdev	average	stdev	average	stdev	average	stdev
KTNS	1728	135,8	4170	177,5	11740	409,4	50860	1563
KTWL	1723	121,3	3946	106,8	10670	284,3	45300	1887
Hybrid	1725	121,2	3950	99,70	10700	275,1	45200	1686

The test run 3 compares the heuristic solutions to the optimal solutions. It can be seen from Table III that at least for small problem instances the heuristics are very effective. The number of jobs (12) and tool category (med) was chosen according to the test run 1. It can be seen from Table III that all the heuristics find near optimal solutions for the small problem instances. The GAPSM3 algorithm is on the average best and its standard deviation is also the smallest.

Table III. Comparison of the optimal and heuristic solutions. The "average" indicates the average cost of 30 problems, "stdev" is the standard deviation of these costs, "smallest" and "largest" are the percentages of the deviations from optimal solutions and "Optimal solutions" indicates the number of optimal solutions for the algorithm. The test data were generated with the following parameters: $|J|=12$, $T=2$, $K=2$, $Tools_{job}=1$, $Tool_{life\text{-}times}$~U(7,9) and $Tool_{switch\text{-}times}$~U(1,3).

Algorithm						
	Opt	**GAPSM**	**GAPSM2**	**GAPSM3**	**SPT**	**LS(2)**
Average	409	412	411	411	439	416
Stdev	50.9	50.8	50.6	50.6	53.7	51.2
Smallest difference	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	2,97 %
Largest difference	0.00 %	1,31 %	1,31 %	1,31 %	12,8 %	5,02 %
Optimal solutions	28	13	23	24	0	7

The test run 4 is used to find out the differences between the heuristic algorithms for larger problem instances. From the computational experience of the test run 2, we use the KTWL rule in our tool switching policy. One can see from Table IV that GAPSM2 is the overall winner, although the local search is surprisingly close to it.

Table IV. Comparison of the different heuristic methods for large problem instances. "Average" is the average cost of 30 problems, "stdev" indicates the standard deviation of the costs and "best" tells the number of lowest costs for the algorithm. Local search uses maximum depth k=2 and tool switching policy applies the KTWL-rule in tool removals. The number of iterations for GAPSM1/2/3 is 50000, size of the population size is 400 and the mutation probability is 0.01. The following parameters were used in the generation process of the test data: $|J|$=100, T=20, K=4, $Tools_{job}$~U(1,3), $Tool_{life\text{-}times}$~U(7,12) and $Tool_{switch\text{-}times}$~U(2,4).

	GAPSM	**GAPSM2**	**GAPSM3**	**SPT**	**LS(2)**
Average	38980	32550	32580	44280	32620
Stdev	1660	1280	1290	1760	1240
Best	0	12	10	0	8

7. Conclusions

A scheduling problem with wearing tools was considered. The previous studies on this subject have either omitted the tool switches due to tool wear completely or concentrated on a single tool problem. The problem definition (SWT) of present research includes cases where there are multiple tools, every job can use a set of different tools and the capacity of the magazine for storing the tools in the processing machine is limited. This is a generalization of the problem presented by Akturk *et al* (2002) and it is also a NP-hard problem.

An optimal solution in a form of an integer formulation was given. The number of decision variables in this model is $5|T||J|+|J|^2$, where T is the set of tools and J is the set of jobs. It was observed that problem instances of practical size were unsolvable within reasonable time limits. Our model can, however, be used for smaller problem instances to obtain an idea of how close our heuristic solutions are to optimal solutions.

Several heuristics to the SWT-problem were introduced. The k-optimal local search is a very simple and fast hill-climbing algorithm which makes only improving moves to the job sequence. Initial job sequence can be obtained, for example, by using the shortest processing time first –rule. The GAPSMx-algorithms are more complex and also more efficient, and they are based on the idea of the “problem space search”. On a coarse level these algorithms are quite similar to the one designed by Akturk *et al* (2002). The main difference lies in the cost evaluation function which must now consider the magazine capacity and the multiple tools used by a job. From empirical testing it was observed that the best algorithm of these was for small problem instances always within 1.3% from the optimal solution cost and on the average 0.11%.

In addition to solution algorithms, two lower bounding methods were considered. The first one is based on linear relaxation. The other lower bound, based on the Lagrange relaxation of the integer formulation did not give significant improvements in running time.

The SWT-problem could still be generalized. One possibility were to allow tools of different sizes. In some applications, like PCB assembly, it is common that one tool may require several positions from the magazine. In these cases the capacity is calculated as slots instead of number of tools it can hold.

Because of the limited capacity of the tool magazine it is possible in this problem formulation that a tool is removed from the magazine before it is completely worn out. This raises the practical issue of how to deal with old tools which could still be used for some jobs. The formulation of the problem allows adding used tools, but our heuristics and the MILP formulation add only unused tools because the cost function does not include the tool usage costs. If these are included, one must either use multiobjective optimization or use single objective optimization with several different factors. This would also require that costs are assigned for all tools.

Two important assumptions were made in this paper: deterministic tool-life and uninterrupted processing of jobs. Interruption between different tools could be allowed but the constant “switching event” cost was considered to be so high that it would not be beneficial. It would also complicate the problem definition and thus make it even harder to solve. Further research on problems with stochastic tool life-times and interruptions are planned.

Further, it was assumed that the tool magazine is empty at the beginning of the scheduling. In dynamic job scheduling this needs not to be the case but the scheduling will be done on the fly from a situation where a number of jobs has been already processed.

Appendix A. Job sequence cost evaluation algorithm SWTC

```
SWTC(jobs, K, R) : totaltime
    totaltime:=0                    -- average time
    usedtime:=0                     -- processing time so far
    M:=$\emptyset$                  -- setting tool magazine empty
    For all $i \in T$ do
        $\alpha(i)$:=0              -- setting life-times of the tools to zero
    End for

    For t=1 to length(jobs) do
        j:=jobs(t)
        For all $i \in T_j$ do
            If  $i \notin M$ then
                If $|M| \geq K$ then   -- if magazine is full then apply removal rule
                    let r:=chooseToolToBeRemoved(jobs, R, M, $\alpha$, t)
                    M:=M \ {r}
                    $\alpha(r)$:=0
                    usedtime:=usedtime+$c_r$
                end if
                M:=M $\cup$ {i}
                $\alpha(i)$:=$l_i$
                usedtime:=usedtime+$c_i$
            else if  $\alpha(i)<p_{j,i}$ then        -- otherwise add the tool
                usedtime:=usedtime+2*$c_i$
                $\alpha(i)$:=$l_i$
            end if
        End for
        totaltime:=totaltime+usedtime
        usedtime:=usedtime+$p_j$
    End for
```

```
chooseToolToBeRemoved(jobs, R, M, α, t): tool

    j:=jobs(t); $best_i$:=null; $best_v$:=0
    For all $i \in M$ do
        If $i \notin T_j$ then                -- checking for tools which can be removed
            $d_{ktns}$ := -1
            $r_i$:= α(i)                      -- remaining life time of tool i
            $t_c$:=t                          -- current time index
            while $r_i$>0 and $t_c \leq$length(jobs) do
                e:=jobs($t_c$)                -- the job of the current time index
                if $i \in T_e$ then
                    if $d_{ktns}$=-1 then $d_{ktns}$:=$t_c$-t
                    $r_i$:=$r_i - p_{j,i}$
                end if
                $t_c$:=$t_c$+1
            end while
            $d_w$:=$t_c - t$

            if R==KTNS then dis:=$d_{ktns}$
            else if R==KTWL then dis:=$d_w$
            else
                dis:=(factor*$d_{ktns}$ + (1-factor)*$d_w$) / $c_t$        -- hybrid rule
            if dis>$best_v$ then $best_v$:=dis; $best_i$:=i
        end if
        return $best_i$
```

Appendix B. The generation parameters for the test data of test run 1

Number of jobs	**Size of the tool set**	**Capacity**	**Number of tools per job**	**Job processing times**	**Tool life-times**	**Tool switching times**
12 low,	1	1	1	U(3,7)	U(7,9)	U(1,3)
med,	2	2	“”	“”	“”	“”
high	4	4	“”	“”	“”	“”
16 low,	1	1	1	U(3,6)	U(7,9)	U(1,3)
med,	3	3	U(1,2)	“”	“”	“”
high	6	4	U(1,2)	“”	“”	“”
20 low,	2	2	U(1,2)	U(3,6)	U(7,9)	U(1,3)
med,	4	4	“”	“”	“”	U(1,4)
high	8	4	“”	“”	“”	U(1,4)

References

Akturk, M.S., Ghosh, J.B. and Gunes, E.D, Scheduling with tool changes to minimize total completion time: A study of heuristics and their performance. *Naval Research Logistics*, 2002, 50, 15-30.

Akturk, M.S. and Avci, S., Tool allocation and machining conditions optimization for CNC machines. *European Journal of Operational Research*, 1996, 94, 335-348.

Bard, J.F., A heuristic for minimizing the number of tool switches on a flexible machine. *IIE Transactions*, 1988, 20(4), 382-391.

Brassard, G. and Bratley, P., *Fundementals of algorithmics*, pp. 205-207, 1996 (Prentice Hall).

Chen, B., Hassin, R. and Tzur, M., Allocation of bandwidth and storage. *IIE Transactions*, 2002, 34, 501-507.

Crama, Y., and Klundert, J., The approximability of tool management problems, 1996, Technical Report rm96034, Maastricht Economic Research School on Technology and Organisation.

Crama, Y., Kolen, A.W. J., Oerlemans, A.G. and Spieksma, F.C.R., Minimizing the number of tool switches on a flexible machine. *The International Journal of Flexible Manufacturing Systems*, 1994, 6, 33-53.

Gray, E., Seidmann, A. and Stecke, K.E., A synthesis of decision models for tool management in automated manufacturing. *Manage Sci*, 1993, 39, 549-567.

Knuutila, T., Hirvikorpi, M., Johnsson, M. and Nevalainen, O.S., Grouping PCB assembly jobs with typed component feeder units. *International Journal of Flexible Manufacturing*, 2004, 16(2).

Hertz, A., Laporte, G., Mittaz, M. And Stecke, K.E., Heuristics for minimizing tool switches when scheduling parts types on a flexible machine. *IIE Transactions*, 1998, 300, 689-694.

Hirvikorpi, M., Knuutila, T., Johnsson, M. and Nevalainen, O.S., Grouping of PCB assembly jobs in the case of flexible feeder units. *Engineering Optimization*, 2005, 37(1), 29-48.

Hirvikorpi, M., Salonen, K., Knuutila, T. and Nevalainen, O.S., General two level storage management problem –reconsideration of the KTNS-rule. To appear in *European Journal of Operational Research*, 2005.

Matzliach, B. and Tzur, M., Storage management of items in two levels of availability. *European Journal of Operational Research*, 2000,121, 363-379.

Mitchell, T. M., *Machine learning*, pp. 249-270, 1997 (McGraw-Hill).

Reeves, C.R., *Modern heuristic techniques for combinatorial problems*, 1995 (McGraw-Hill International).

Smed, J., *Production planning in printed circuit board assembly*, PhD thesis, TUCS Dissertations 36, University of Turku, 2002.

Storer, R.H., Wu, D.S and Vaccari, R., New search spaces for sequencing problems with application to job shop scheduling. *Manage Sci*, 1983, 29, 273-288.

Tang, C.S. and Denardo, E.V., Models arising from a flexible manufacturing machine. *Operations Research*, 1988, 36(5), 767-84.

Qi, X., Chen, T. and Tu, F., Scheduling the maintenance on a single machine. *Journal of Operations Research Society*, 1999, 50, 1071-1078.

Publication VI

Hirvikorpi M., Knuutila T., Leipälä T. and Nevalainen O.S., Job Scheduling and Management of Wearing Tools with Stochastic Lifetimes, Submitted for publication, 2004.

Job Scheduling and Management of Wearing Tools with Stochastic Lifetimes

Mika Hirvikorpi [1,*]
Timo Knuutila [1]
Timo Leipälä [2]
Olli S. Nevalainen [1]

1) Department of Information Technology, University of Turku and
Turku Centre for Computer Science (TUCS)
Lemminkäisenkatu 14 A 20520 Turku
2) Department of Mathematical Sciences, University of Turku, 20014 Turku

[*] corresponding author
mika.hirvikorpi@utu.fi
phone: +358-2-3338690
fax: +358-2-3338600

This work has been partially financed by the
Graduate School, Turku Centre for Computer Science (TUCS)

Abstract

The problem of scheduling jobs using several wearing tools is studied. The lifetimes of the tools are stochastic and one flexible manufacturing machine provided with a limited capacity tool magazine processes all the jobs. Problem here is to minimize the expected average completion time of the jobs by choosing the processing order of the jobs and tool management decisions wisely. This kind of situation is met in planning the production control of CNC-machines.

Previous studies concerning this problem have either assumed deterministic lifetimes for tools or omitted the wearing of tools. In our formulation of the problem, tool lifetimes are supposed to be stochastic and to follow some known probability distribution.

We give a mathematical formulation for the problem. Lower and upper bound methods are introduced along with a genetic algorithm solving the problem heuristically. Empirical tests indicate that when the stochastic information is taken into account, we can reduce the average job processing time considerably.

Keywords: CNC production, job scheduling, tool wear, stochastic tool lifetimes, heuristic algorithms

1. Introduction

This paper considers the problem of *Job Scheduling with Stochastic Tool Lifetimes* (JSSTL). JSSTL-problem arises in an environment which consists of a single manufacturing machine using wearing tools to process some *jobs* (called also *parts*). The job set is known in advance and the task is to select an order of the jobs and make such tool management decisions that the expected average processing time of the jobs is minimized. While the order of processing the jobs is free, it is supposed that the order of using different tools on a given job is specific to the particular job. The later assumption refers to a situation where the precedence of the processing steps with different tools defines a total ordering. The JSSTL-problem is stochastic in the sense that the lifetimes of the tools follow known tool specific probability distributions. Tools can be switched only between jobs and the number of tools the machine can use is limited by the capacity of the *tool magazine*. Tools which are not in the magazine are kept in a secondary storage in the vicinity of the machine. A time cost is associated with each *tool switch*. If a tool is broken during the processing of a job, the unfinished job is discarded and it is processed again from the beginning. It is assumed that each time a tool is loaded in the tool magazine it is “new”, i.e. no partially worn tools are reloaded.

Tool management literature considers tool switching occurring due to the job mix. The objective there is to minimize the *total number of tool switches* (for a fixed order of jobs) or to minimize the *number of switching instants* by grouping the jobs, see Knuutila et al. (2001). For infinite tool lifetimes the KTNS-rule developed by Tang and Denardo (1988) minimizes the number of tool switches when the order of jobs is fixed and the capacity demands of the tool magazine space for each tool are uniform, this was shown by Crama et al. (1994). The study of Gray et al. (1993) shows, however, that tool switching occurring due to the tool wear may be ten times more frequent than one occurring due to the job mix.

In the scheduling literature the tool switching is usually omitted completely. The simplest form of scheduling is the case where a set of jobs with certain fixed processing times is given and task is to choose such a processing order that the average completion time of the jobs is minimal. This problem can be solved for the single machine case exactly by using the *shortest-processing-time-order* (SPT-rule). A more complex scheduling model including machine breakdowns with stochastic repair times was studied by Adiri et al. (1989). They proved that the SPT-rule minimizes the expected total flow time if the breakdown times are negative exponentially distributed. This same model with deterministic repair times was also considered by Lee and Liman (1992), who proved that the SPT-order is always within factor of 9/7 of the optimal flow time. Two-machine models has been studied by Lee (1996,1997). Common to these models is the assumption that breakdown or maintenance occurs only once during the processing of the jobs.

Qi et al. (1999) consider a single machine scheduling problem with multiple maintenance intervals of variable duration. Their model is equivalent to the one considered by Akturk et al. (2002) who seem to be the first authors to study the tool wear in scheduling. Their model allows the use of one tool type which has a deterministic lifetime. Although quite simple, this kind of formulation is relevant because it is not unusual that only one tool is used to process several different jobs, for example in a cutting application. A more general formulation of the wearing tools problem was given by Hirvikorpi et al. (2003) who allowed the use of several tool types and supposed that the capacity of the tool magazine is limited. This formulation assumed deterministic lifetimes and equal sizes for the tools. The present paper extends the model of Hirvikorpi et al. (2003) by relaxing the assumption on deterministic lifetimes but still keeping the equal widths. This brings us to a situation where the lifetimes of the tools are stochastic with known probability density functions. A natural question then rises how to plan the tool management and processing order in such a way that the processing of the jobs is as efficient as possible (in the sense of the expected average completion time of the jobs). A good tool management plan should avoid breakdowns of tools under operation but also excessive tool changes.

Taylor’s tool-life formula is a traditional way of calculating the lifetime of a tool when certain parameters like cutting speed and cutting depth are given. The formulation of the JSSTL-problem (see section 2) assumes that these parameters are fixed before the processing. Although this makes the model less accurate for certain types of tools, it allows us to use any kind of tool for which the lifetime probability distribution is known. Using Taylor’s formula in the formulation would limit it’s use.

The presentation is organized as follows. Section 2 defines the *Job Scheduling with Stochastic Tool Lifetimes* –problem. Section 3 considers two different lower bounding methods and calculation of an upper is also discussed. Section 4 introduces a heuristic cost evaluation algorithm when the order of jobs is predefined. This algorithm is used to guide the search process in the genetic algorithm (GA) designed for solving the joint problem of scheduling and tool management, as presented later in the same section. Section 5 presents empirical results computed using the lower bound methods and our GA. The paper is concluded in section 6.

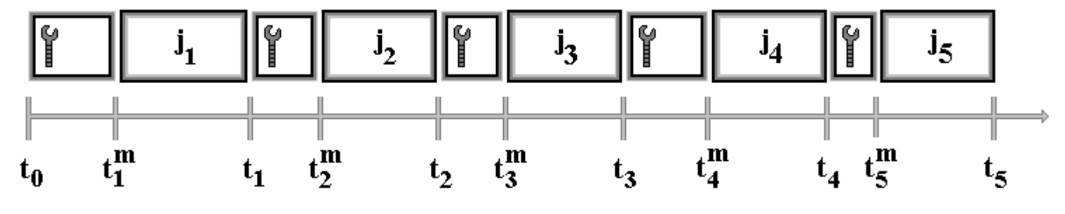
2. Definition of the JSSTL-problem

In the *Job Scheduling with Stochastic Tool Lifetimes* –problem (JSSTL) a set of jobs J to be processed with a single machine is given. The jobs are processed with one or several tools each, and the time for using a tool depends on the tool-job combination. The tools are used serially in a fixed order specific to each job. The machine is able to use only the tools which are in its tool magazine at the moment of starting the processing of the job. This magazine is of limited capacity. Each tool reserves the same amount of the capacity of the magazine and the time for changing a tool depends on the tool type, only. It is further assumed that the magazine can hold only a subset of the tools all the jobs require in whole. The lifetimes of the tools follow some known probability distributions. If a tool breaks down during the processing, the unfinished job is discarded and it is processed from begin again with (at least partially) renewed tool collection. The problem itself is to choose such a processing order of the jobs and tool management decisions that the *expected average completion time of jobs is minimized*. This formulation does therefore put no extra cost to jobs during which a tool has broken. In the following presentation the concepts *cost* and *time* are used interchangeably.

2.1 Basic concepts

While the main idea of solving the JSSTL-problem is simple, the formulation needs definition of new concepts. *Tool realization* stands for one individual tool with a certain lifetime. When the processing of the jobs begins there is some set ξ of tool realizations. For this set, $\xi(i,l)\in \mathrm{N}$ stands for the lifetime of the l:th realization of tool i. For the sake of simplicity this definition assumes that there is a potentially unlimited number of realizations of each tool the processing requires.

Two more concepts are *linear time (cost)* and *quadratic time (cost)* denoted by c_l and c_q, see Fig. 1. Assume that a set of jobs J has been processed in some order. The linear time is the sum of realized processing times of all jobs in J. This time is stochastic and depends on the particular tool realizations observed when processing the jobs. Note that a particular job may need several realizations of a certain tool due to the possible tool break-downs. Quadratic time is the sum of completion times of the jobs in J. Dividing the quadratic time with the number of jobs ($|J|$) gives the *average completion time of the jobs* of J for one set of tool life realizations. The objective in the JSSTL-problem is to minimize this average completion time of the jobs.



$$c_l = t_5$$

$$c_q = \sum_{i=1}^{5}(5-i+1)t_i + \sum_{i=1}^{5}(5-i+1)(t_i^m - t_{i-1})$$

Figure 1. Illustration of the concepts of *linear time* c_l and *quadratic time* c_q when processing the jobs $J=\{j_1, j_2, j_3, j_4, j_5\}$. There is a tool management interval of variable duration between each job.

2.2 A two-phase stochastic model for the JSSTL-problem

We use a two-phase stochastic model for defining the JSSTL-problem. In the first phase, we choose the processing order of the jobs, the *magazine states* (see below) and the reload decisions of the tools between jobs. In the second phase, a cost function is used to calculate the incurred cost for all possible realization sets of the tools. These costs are weighted with their probabilities and then summed up to form the average cost under the condition that the decisions of the first phase have been made. The difficulty in defining even the cost function for a stochastic problem like this is the possibility of an infinite sequence of events (break-downs). This is why recursion is needed in the calculation of the costs.

2.2.1 Problem instance

The following parameters describe an instance of the JSSTL-problem:

T *set of tools*,

- c_i the time required to remove (load) tool $i \in T$ from (to) the tool magazine (this is called the *handling cost* of tool i; the cost does not depend on the current allocation of tools in the magazine).

J *set of jobs*,

- T_j the set of tools required by job $j \in J$, $T_j \subseteq T$,
- $T_{j,n}$ the n:th tool required by job j (a fixed ordering of using tools is supposed for each job),
- $p_{j,n}$ the processing time of the n:th tool for job j.

K *capacity* of the tool magazine (the number of tools it can hold simultaneously; each tool consumes a single "slot" of the magazine),

2.2.2 Operation plan

In order to define an *operation plan* which describes a solution to a JSSTL-problem instance, the concept of process state needs to be defined:

S	*process state* S is a three-tuple (M, R, j), where j is the job to be processed next, M is the set of tools currently in the magazine and $R \subseteq M$ is the set of tools which have been reloaded just before processing job j. The projection function $j(S)$ returns the job of the process state, $M(S)$ returns the tool set currently in the magazine and $R(S)$ gives the reload set.

Process state $S=(M, R, j)$ is said to be *valid* if and only if $T_j \subseteq M$ and *feasible* with respect to capacity K if $|M| \leq K$.

An *operation plan* A is defined to be a sequence of process states $A=(S_1, S_2, \ldots, S_{|J|})$. It thus fixes the tool management decisions and the processing order of the jobs.

We say that an operation plan $A=(S_1, S_2, \ldots, S_{|J|})$ is *legal* if

1) all the process states S_k, $k=1,\ldots,|J|$ are valid and legal with respect to the capacity K and
2) $\bigcup_{k=1}^{|J|} \{ j(S_k) \} = J$.

2.2.3 Tool management sets and events

Tool usage set U consists of three-tuples (i, r, u) where $i \in T$, $r \in \mathrm{N}$ and $u \in \mathrm{R}$. The tuple describes the consumed time u for n:th realization of tool i. A tool usage set keeps record of how the tools are used during the processing. In addition, we define the following functions: $\beta(U, i)$ returns the latest realization index of tool i found from set U and $\alpha(U, i)$ gives the consumed time for the lastest realization of tool i.

We must define two events before we can give a cost function for processing the jobs. Assume that job j is currently processed with the n:th required tool, the current tool usage set is U (giving the consumed times of tools for j) and the set of tool realizations is ξ (the realized lifetimes $\xi(i, l)$ are drawn from the lifetime distributions).

Then the *Tool process event* is defined as follows:

if $\alpha(T_{j,n}, U)+p_{j,n} \leq \xi(T_{j,n}, \beta(T_{j,n}, U))$ then the new tool usage set is $U \cup \{s_{ok}\}$, where $s_o=(T_{j,n}, \beta(T_{j,n}, U), \alpha(T_{j,n}, U)+p_{j,n})$.

This event stands for the case of successful processing of the job j by tool $T_{j,n}$ in which case the tool usage set U is augmented by a new element s_{ok} for $T_{j,n}$.

Tool reloading event is defined as follows:

if $n \leq |T_j|$ and $\alpha(T_{j,n})+p_{j,n} \geq \xi(T_{j,n}, \beta(T_{j,n}, U))$ then the new tool usage set is $U \cup \{s_{fail}\}$, where $s_{fail}=(T_{j,n}, \beta(T_{j,n}, U)+1, 0)$.

This event occurs when the current realization of tool $T_{j,n}$ is not able to process job j, i.e. $T_{j,n}$ breaks while it is used.

2.2.4 Cost functions

We can now formulate *job processing cost function* $P_c(j,U,\xi,n,c)$. Function P_c calculates a 2-tuple consisting of the linear cost c of processing job j and the (updated) tool usage set U. The input parameters for P_c are:

j	job to be processed,
U	current tool usage set,
ξ	set of tool realizations,
n	current tool index used to process job j and
c	the incurred linear cost from the previous steps.

Function P_c is defined as follows:

$$P_c(j,U,\xi,n,c) = \begin{cases} P_c(j,U \cup \{s_{0k}\},\xi,n+1,c+p_{j,n}) & \text{if } n \le |T_j| \text{ and a tool process event occurs} \\ P_c(j,U \cup \{s_{fail}\},\xi,1,c+2c(T_{j,n})) & \text{if } n \le |T_j| \text{ and a tool reloading event occurs} \\ (c,U) & \text{if } n > |T_j| \end{cases}$$

In the first case the job j is processed with n:th tool and the current realization of this tool is able to do its task, this cornensponds to the tool process event defined above. In the second case the situation is opposite to the above, the current realization of the n:th tool required by the job j is not able to process the job and a tool reload event occurs. This incurs twice the handling cost (i.e. remove and then load a new realization) of the broken tool and job j must be restarted with tool 1 again. In the last case job j has been processed with all tools it requires and the processing of job j is ready. Function P_c then returns the incurred linear cost c and the tool usage set U after processing job j.

The procedure $PLAN_c$ calculates the average job processing cost of a given operation plan A and tool-life realization ξ. The general form of the procedure is:

$PLAN_c(A, k, \xi, c_q, c_l, U)$.

Parameter A is the operation plan we are calculating the cost for, k is the index of the current job in the operation plan, c_q is the quadratic time incurred, c_l is the linear time incurred so far and U tells the state of the magazine. Now we can define the total cost of the operation plan A for the tool realization ξ:

```
$PLAN_c(A,\xi)$ : returns integer
    return $PLAN_c(A, 1, \xi, 0, 0, \{\})$;
```

This first part of the definition is a wrapper function, simplifying the notation to be presented later. The general case of the procedure $PLAN_c$ is defined as follows:

```
$PLAN_c(A, k, \xi, c_q, c_l, U)$ : returns integer

    Let $(M_{curr}, R_{curr}, j_{curr}) = A(k)$
```

```
$U_{start} = U \cup \{(i, \beta(U,i)+1, 0) \mid i \in R_{curr}\}$;         -- $U_{start}$ is the new magazine state after
                                                                           -- tool reloading has been made
$c_{reload} = 2\, c(R_{curr})$;                                            -- cost of replacing the tools in this state
$(c_{process}, U_{processed}) = P_c(j_{curr}, U_{start}, \xi, 1, 0)$;      -- calculating the cost of processing $j_{curr}$
                                                                           -- and the new magazine state
$c_{finish} = c_l + c_{reload} + c_{process}$;                             -- finishing time of the job $j_{curr}$

if k<|J| then

    Let $(M_{next}, R_{next}, j_{next}) = A(k+1)$;                         -- $T_{remove}$ is the set of tools to be
    $T_{remove} = M_{next} \setminus M_{curr}$;                            -- removed when moving to the next
                                                                           -- state
    $U_{ready} = U_{processed} \cup$                                       -- updating the tool realizations for the
    $\{(i, \beta(U_{processed}, i)+1, 0) \mid i \in T_{remove}\}$;         -- removed tools

    $c_{state} = c_{reload} + c_{process} + c(T_{remove})$;               -- cost of the state
    return $PLAN_c(A, k+1, \xi, c_q + c_{finish},$                         -- calculating the cost for the rest of
                $c_l + c_{state}, U_{ready})$                              -- the operation plan
else
    return $(c_q + c_{finish})/|J|$                                        -- returning the average cost
```

2.2.5 Definition of the JSSTL-problem

The problem of *Job Scheduling with Stochastic Tool Lifetimes* (JSSTL) is defined as follows: for given capacity K, job set J and tool set T, find a legal operation plan A for which

$$E_\xi(PLAN_c(A,\xi)) = \min! \qquad (1)$$

The operator E_ξ denotes the expected value over all tool realizations (ξ) drawn from the given tool-life distributions.

2.2.6 An example

Consider a JSSTL-problem instance consists of 4 jobs to be processed with 4 tools, see Fig. 2. Between each job the operation plan describes tool loadings (arrow down), removals (arrow up) and reloadings (two-headed arrow). The numbers in the figure indicate which tool is in question, for example job 1 is processed with tools 1 and 2, in this order. The tool magazine states are shown below the processing order. The height of a bar in tool magazine indicates the remaining lifetime of a tool. The capacity of the tool magazine is 2. No tool break-downs are shown in this simple example.

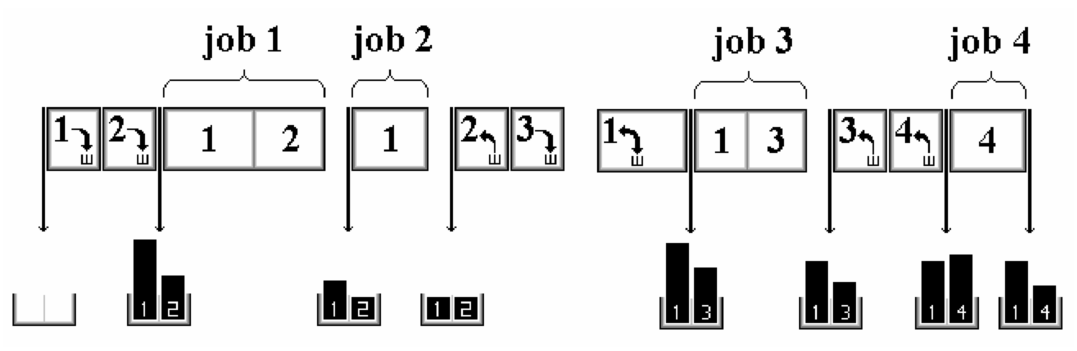


Figure 2. An example of an operation plan with 4 tools, 4 jobs and magazine capacity of 2 tools.

3. Bounding methods

One can use the MILP model of Hirvikorpi et al. (2003) (see also Appendix A) to calculate lower bounds for JSSTL-problem instances if certain assumptions about the lifetimes of the tools are made. The additional assumption is that the probability of a breakdown for each tool $i \in T$ is 1 for lifetimes greater than a fixed finite time max_i. Otherwise, it is possible that some tool never breaks down and the lower bound would not hold. With this assumption the lower bound LB_{max} can be calculated by using the model of Hirvikorpi et al. (2003).

Another possibility of calculating lower bounds for the problem is to assume infinite lifetimes for tools. The benefit of this approach is that we need not worry about the probability distributions of the tool-lifes. With this assumption the following model can be used to solve lower bounds of JSSTL-problem instances. As in section 2, let c_i , K, p_j and (i=1,2,…,T and j,t=1,2,…,P) stand for the tool handling costs, the capacity of the tool magazine and the total processing times of the jobs, respectively. In addition, let define $h_{j,i}$ to be 1 if job j requires tool i and 0 otherwise.

The decision variables are:

$x_{j,t}$	1, if job j is processed as the t:th in the sequence, 0 otherwise;
$z_{i,t}$	1, if tool i is in the magazine when processing the t:th job in the sequence, 0 otherwise;
$s_{i,t}$	1, if tool i is removed or added before beginning the processing of the t:th job, 0 otherwise.

The LB_{inf} -problem can then be stated as:

minimize

$$\sum_{t=1,\ldots,T} (p-t+1)(\sum_{j=1,\ldots,P} x_{j,t}\, p_j + \sum_{i=1,\ldots,T} s_{i,t}\, c_i) \tag{2}$$

subject to

$$\sum_{j=1,\ldots,P} x_{j,t} = 1 \tag{3}$$
$$\sum_{t=1,\ldots,P} x_{j,t} = 1 \tag{4}$$
$$z_{i,t} \geq \sum_{j=1..P} x_{j,t}\, h_{j,i} \tag{5}$$
$$s_{i,t} \geq z_{i,t} - z_{i,t-1} \tag{6}$$
$$s_{i,t} \geq z_{i,t-1} - z_{i,t} \tag{7}$$
$$\sum_{i=1,\ldots,T} z_{i,t} \leq K \tag{8}$$
$$x_{j,t},\ z_{i,t},\ s_{i,t} \in \{0,1\} \tag{9}$$
$$z_{i,0} = 0 \tag{10}$$

The objective function (2) calculates the quadratic processing time. The inner summation of (2) consists of two parts: job processing time and tool management time (see Fig. 1). The constraints (3) and (4) guarantee that every job is processed exactly once. Constraint (5) says that all the tools required during each time period must be in the magazine. The following two constraints guarantee that whenever a tool is added or removed, its decision variable s is set to 1. Constraint (8) says that the given capacity must not be exceeded. Constraint (9) sets the domains for the decision variables and constraint (10) says that the tool magazine is originally empty.

If the occurrence of tool-lifes shorter than the processing times are allowed, then the upper bound for the JSSTL-problem instance is infinite. This is due to the fact that we may have to process some job an infinitely long time due to repeated breakdowns of some tools. One could, however, calculate upper bounds which hold with certain probability, but this seems to be rather complicated. Other possibility would be to assume that each tool $i \in T$ lasts at least a certain minimum time min_i. For all tools, min_i has to be greater than or equal to the maximal processing time of tool i among all jobs, then each job can be processed within the lifetime of one tool. With this assumption the upper bound UB_{min} is calculated by using the model of Hirvikorpi et al. (2003) and assuming lifetimes min_i for all tools.

4. Heuristics for the JSSTL-problem

The JSSTL-problem can be solved heuristically by using local search methods, see Reeves (1995). The solution presented here uses a genetic algorithm (GA) which is guided by a deterministic or stochastic cost evaluation algorithm (DCEA, SCEA). DCEA makes the tool management decisions by using the average lifetimes for the tools. SCEA exploits the stochastic information in the tool management decisions. Both of these cost evaluation algorithms are based on simulation.

4.1 Cost evaluation based on the average lifetime, DCEA

The method used in the deterministic cost evaluation algorithm is similar to the one described by Hirvikorpi et al. (2003). The algorithm proceeds iteratively through the tool requests. During each iteration DCEA first makes some tool management decisions and then determines the processing cost of the current job using simulation.

The tool management decisions DCEA must make are loading, removal and reloading decisions. Tool loadings are made if some of the tools the current job requires are not in the magazine. If the magazine runs out of the capacity when loading the tools, the algorithm must choose some of the tools to be removed. These tools are selected by using the removal rules introduced by Hirvikorpi et al. (2003). The KTNS-rule (Keep Tool Needed Soonest) chooses the tool which is needed latest. The KTWL (Keep Tool which Wears out Last) removes the tool which wears out first and the HYBRID calculates a weighted sum of KTNS and KTWL. Tools which are already in the magazine and are needed by the next job are reloaded if their remaining lifetime is shorter than the next job requires. Because the tool lifetimes are stochastic the algorithm uses the expected lifetimes of the tools when making the decisions.

Once DCEA has made the necessary tool magament decisions, it simulates the processing of the current job. Because the real lifetimes for the tools are not known a tool may break during the processing. In this case the DCEA discards the unfinished job, reloads the broken tool and processes the job from the beginning again. This process is iterated as many times as necessary. When the current job has been processed successfully it moves to the next job.

The pseudocode of DCEA is in Fig. 3. The algorithm uses as its input *J*, the ordered set of jobs, *K*, the capacity of the tool magazine and *R*, the rule for tool removals (KTNS/ KTWL/ HYBRID). One should notice that in practice it is advisable to run the cost evaluation several times and calculate the average cost of these runs in order to get a better approximation for the expected cost of the given order. This is because each run calculates the cost of one realization set of tool lifetimes, only.

```
DCEA(J, K, R) : average cost
    let quadratictime = 0;
    let lineartime = 0;
    For all jobs j in order given in J
        if tools needed by j are not in the magazine then
            load the required tools and use the removal rule R  if necessary;
            add the tool management time to lineartime;
        reload the tools which are too worn out (i.e. can not handle the job j when
        measured with the expected lifetime) and add the time required to reload them to
        lineartime;

        quadratictime = quadratictime + lineartime;

        for all tools i required by j
            if tool i breaks down during processing then
                reload the tool and begin the processing of j from the begin;
            else
                add the processing time with tool i to lineartime;
    return quadratictime /|J|
```

Figure 3. Deterministic cost evaluation algorithm.

4.2 Stochastic cost evaluation, SCEA

The main phases of the stochastic cost evaluation algorithm (SCEA) are the same as for DCEA. However, when making the tool management decisions, SCEA takes the advantage of the stochastic information (i.e. tool lifetime distributions) which is supposed to be known.

For the removal decisions similar rules as for the deterministic case apply. The DCEA used the expected lifetime of tools when making the removal decisions. Using the expected lifetime means that certain percentage of the tools break down before reaching this age. Additional notation is needed to describe the stochastic rules:

$P_i(l)$	the probability of tool *i* that the lifetime is less than equal to *l*,
$l_i(u)$	the lifetime of the tool *i* for which the theoretical breakdown probability is greater than or equal to *u* (i.e. $l_i(u)= P_i^{-1}(u)$).

Instead of using the expected lifetimes in the removal decisions, we parametrize the rules with the *u*-value and use the corresponding lifetime $l_i(u)$ when making the tool management decisions. The *u*-value is one of the parameters of the SCEA and it enables the algorithm user to control the risk of tool breakdowns.

A greedy approach in minimizing the cost of a process plan would be to make such reload decisions that the expected cost of processing the *next* job *j* is minimized. Let us next formulate the expected cost of processing *j* when the state of the tools in the magazine is known. This formulation requires two additional notations. Let

$IP_i(t, d)$	be the probability that tool $i \in T$ breaks down during the time interval $[t,t+d]$.

The *expected cost* $EC(j,U)$ of processing job *j* when the current status of tools is known through *U* is:

$$EC(j, U) = \overline{EC}\ (j, U, 1),$$

where

$$\overline{EC} = \begin{cases} 0 & \text{if } n > |T_j| \\ IP_{T_{j,n}}(\alpha(T_{j,n},U), p_{j,n}) * (2c(T_{j,n}) + \overline{EC}(j, U \cup \{s_f\}, 1)) + & \text{if } n \le |T_J| \\ (1 - IP_{T_{j,n}}(\alpha(T_{j,n},U), p_{j,n})) * (p_{j,n} + \overline{EC}(j, U \cup \{s_0\}, n+1)) & \end{cases}$$

The recursive function $\overline{EC}$ stands for the expected remaining cost for job *j* when processing it with tools $T_{j,n}, T_{j,n+1}, \ldots, T_{j,|T_j|}$. In the first case of $\overline{EC}$ the recursion ends because job *j* has been processed with all tools it requires. In the second case there are two possible events that can occur: tool process event or tool reloading event (see section 2) . Function $\overline{EC}$ calculates recursively the expected cost for both of these events and weights them with their probabilities.

The only way of minimizing the value of $\overline{EC}$ for given *j* and *U* is to change the initial state of tools *U*. More precisely, one has to decide which tools are reloaded. A brute force approach would be to try all combinations of reloadings and choose the one with the lowest expected cost. There are, however, exponential number of such combinations with respect to the number of tools required by *j*. Furthermore, the exact evaluation of $\overline{EC}$ leads to infinite recursion. Exact evaluation is however not needed if an approximate of its value is satisfactory. For an approximate value the depth of the recursion can be limited by setting a probability limit parameter p_{cut} which cuts the recursion when the probability of the recursion path drops below this level.

Our heuristic for tool reloads (SCEA) calculates for each tool *i* in the magazine the gain (i.e. the decrease) in the expected cost if *i* is reloaded. It then chooses the reloading for which the gain is largest. Once the reloading has been done the reloading gains are evaluated again and another selection is made. This is repeated as long as the expected cost decreases. Figure 4 shows SCEA on a coarse level. Two additional parameters, when compared to DCEA are given, the *u*-value and p_{cut}. Through the *u*-value we can control the risk of tool breakdowns during the processing and the p_{cut} limits the recursion when evaluating $\overline{EC}$ as described before. The *u*-value is given as a parameter to the removal rules (marked with `R(u)` in the pseudocode) .

```
SCEA(J, K, R, u, $p_{cut}$) : average cost
    let quadratictime = 0;
    let lineartime = 0;
    For all jobs j in order given in J

        if tools needed by j are not in the magazine then
            load the required tools and use the removal rule R(u) if necessary;
            add the time required to lineartime;
        while the expected cost decreases
            calculate the change in the expected cost for all tools not yet reloaded;
            if the largest change is positive then
                reload the tool and add the reloading time to lineartime;

        quadratictime = quadratictime + lineartime;

        for all tools i required by j
            if tool i breaks down during processing job j then
                reload the tool and restart the processing of j from the beginning;
                add the handling time of the broken tool and the time used before
                breakdown to lineartime;
            else
                add the processing time with tool i  to lineartime;

    return quadratictime /|J|
```

Figure 4. Stochastic cost evaluation algorithm.

4.3 A genetic algorithm for job scheduling, SSA

The *stochastic scheduler algorithm* (SSA) described in this section is a modification of the genetic algorithm GAPSM by Hirvikorpi et al. (2003). The GAPSM was developed for the deterministic version of this problem. For a general description of a genetic algorithm (GA) and for the description of the original GAPS algorithm, see Reeves (1995) and Akturk et al. (2002).

SSA has many technical details which are not of general interest, therefore only an overall description of the algorithm is given here. The key components of a GA include coding an individual, fitness evaluation, crossover, mutation and selection. These components are discussed next and after that a brief description of the main phases of SSA is given.

The coding of an individual is based on the idea of the problem space search, see Storer et al. (1983). In the JSSTL-problem a set of jobs *J* is given as an input. A single individual of SSA describes some permutation of the jobs in *J*. The individuals are therefore coded so that they can describe all the $|J|!$ possible permutations: an individual is coded as a vector of perturbation factors, which are real numbers from interval (0,1]. Each job in *J* is associated with a factor which is used as a multiplier to the processing time of the job in question. It is now possible to use almost any kind of deterministic base heuristic to calculate an order for the jobs from these perturbed processing times. The heuristic must be deterministic in the sense that when it is given certain processing times, it produces always the same order. SSA uses the SPT-rule which sorts the jobs to an order of increasing processing times. It is now easy to see that by choosing the perturbation factors for the jobs properly, one can generate any of the permutations of the jobs by applying the SPT-rule to these perturbed processing times. In this way, an individual which is an ordering of the jobs is described fully by two components: a perturbation vector and the fixed base heuristic.

The fitness of an individual is evaluated in two steps. In the first step SSA uses the SPT-heuristic (our base heuristic) to create an ordered sequence of the jobs. The processing times which have been perturbed (multiplied) with the perturbation vector of the individual are used in this step. In the second step SSA uses the result of the first step as an input to the cost evaluation algorithm (DCEA or SCEA). The result given by cost evaluation algorithm, which is an approximate of the average processing cost for the given order, is the fitness of the individual. (This is the major difference between GAPSM and SSA, since GAPSM uses a deterministic cost evaluation algorithm due to the different nature of the problem it solves.)

In the crossover, parents are chosen by tournament selection. During each tournament round candidate parents are selected randomly from the population and compared against the current ones. The number of tournament rounds is selected randomly from an interval which is given as parameter. For example, if the number of tournament rounds is 0 then the selection of the parents is purely random. After the selection of the parents, the algorithm performs one-point crossover by taking the first half of the perturbation factors from the other parent and the rest from the other. These halfs are then catenated to form an offspring.

Each individual in the population is allowed to mutate with a small probability during each iteration. When mutation event occurs, the algorithm selects one random position in the perturbation vector of the individual in question and replaces it with a random number from inteval (0,1].

During each iteration of SSA, one offspring is created in a manner described earlier. This offspring then replaces the individual in the current population which has the worst fitness (i.e. highest average processing cost). All the other individuals are directly transferred to the next iteration.

The SSA algorithm begins by creating an initial population of certain size (given as a parameter). This population is created purely random so that for each individual every perturbation factor is chosen from a uniform distribution of (0,1]. In the evolution phase SSA selects the individuals for the next generation through direct transfer and crossover as described. After the selection each individual of the next generation is allowed to go through a mutation with a certain probability (a parameter). The evolution phase is repeated a given number of iterations and the result of the SSA is the order described by the individual with the best fitness after the last iteration.

5. Empirical results

The following test runs are based on synthesized data but the various parameters used in the synthesization process were given by our industrial partners. It is important to realize that the number of tools and jobs vary greatly depending on the size of manufacturing facility and the type of parts processed. For example a manufacturer producing large engines uses a flexible machine for which the capacity of the tool magazine is several hundreds of tools. In contrast to this, small work shops can have machines for which the capacity of the tool magazine is less than 20 tools.

Based on the examples given above we use three different problem sets with 30 instances in each. These sets were created with the following parameters: 20 jobs with 1 tool (called small), 50 jobs with 8 tools (medium) and 100 jobs with 20 tools (large). The tool lifetimes were taken from the uniform distribution U(10,90), the job processing times from U(1,10) and tool loading (removal) times from U(1,5). The number of tools per job for small problem instances is 1, medium problem instances a random number from U(1,2) and large problem instances from U(1,4). The capacities of the tool magazine are 1, 4 and 8. These parameters are used as default in the following test runs.

The evaluation of the expected average processing cost (EAPC) is based on sampling of tool realizations. Each run of DCEA or SCEA calculates the cost of one possible tool realization set and is thus one sample. Averaging these samples gives an approximate of the EAPC for a given order of jobs.

We have two goals in our testing. The primary goal is to evaluate the performance of the SSA. In order do this we must first tune the parameters of the cost evaluation algorithms. The most important parameter is the number of samples used per cost evaluation. This is due to the fact that if the variance of the cost evaluation is too large then the results given by the SSA are not reliable. It is also possible that for variances large enough the cost evaluation algorithm is not able to guide the search to improving directions. The second parameter which needs to be tuned is the *u*-value which controls the risk of tool break downs during the processing. Our secondary goal in testing is to find out what is gained by using the stochastic information in the optimization. One possible way to do this is to compare the results of DCEA and SCEA.

5.1 Estimating the variance

The first set of test runs measures the standard deviation of the cost evaluation algorithms. This is necessary to be able to find out a reasonable number of samples needed per cost evalution for DCEA and SCEA. Another objective is to find out how the problem size affects the standard deviation of the cost evaluation algorithm. From Fig. 5 one can see that the standard deviation diminishes rapidly as a function of the number of samples. As expected, the deviations decrease as the number of jobs gets larger. This is due to the fact that even minor changes in the job schedule have then larger proportional effect to the cost for small problem instances than for large ones. Based on these results 200 samples are used for medium and large problem instances in later test runs. For small problem instances additional test runs for 300, 500 and 1000 samples were made (data not shown). It turned out that the deviation was small enough for our purposes (5.5 % of the mean) when using 1000 samples per cost evaluation.

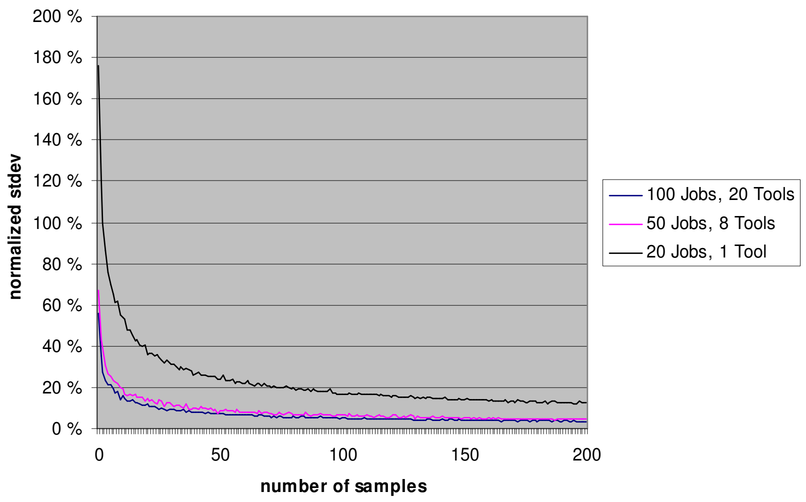
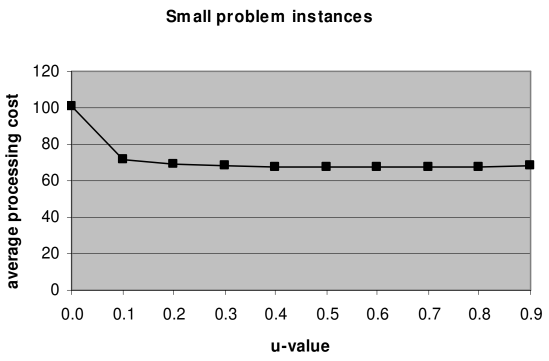


Figure 5. Standard deviation of the average processing cost for SCEA when the number of samples varies between 1 and 200. The standard deviation is calculated by evaluating the cost 30 times for a random order with the given number of samples. The deviation is normalized by the mean of the processing cost. For larger problem instances (e.g. 50 and 100 jobs) the deviation is small enough (5.5% of the mean) when the number of samples is 200. For small problem instances more sampling (1000 samples) is required.

5.2 Tuning the stochastic cost evaluation algorithm (SCEA)

Another important parameter in the cost evaluation is the *u*-value which determines the lifetimes used in the removal rules of SCEA and DCEA. An ideal situation would be a universal *u*-value for all problem sizes.



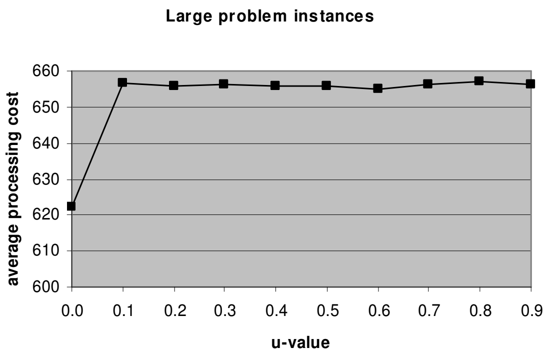


Figure 6. Tuning the *u*-value for the stochastic cost evaluation algorithm (SCEA).

Figure 6 shows the dependence of the average processing cost as a function of the *u*-value. The *u*-value 0.0 is the only one showing difference of statistical importance (pairwise *t*-test, confidence 0.99) for both small and large problem instances when compared to other *u*-values. This difference is caused by tool changes and tool breakdowns in the beginning of the schedule. For small problem instances a too aggressive tool change politic (small *u*-value) causes a larger average processing cost because the tool changes have then a larger proportional effect than in large problem instances. Based on this test we use the *u*-value 0.5 for small problem instances and 0.0 for large problem instances.

5.3 Evaluating performance of the stochastic scheduler algorithm (SSA) performance

The overall performance of SSA is profiled against lower and upper bounds in this test run. The distribution parameters are chosen so that all jobs can be processed even if any tool breaks down as soon as possible. The bounds are evaluated by using the model presented by Hirvikorpi et al. (2003), see Appendix A. Figure 7 summarizes the results of this test. For these small problem instances the SSA works quite efficiently: the lower bound costs and SSA costs have no difference of statistical significance, when tested with paired t-test (confidence 99%).

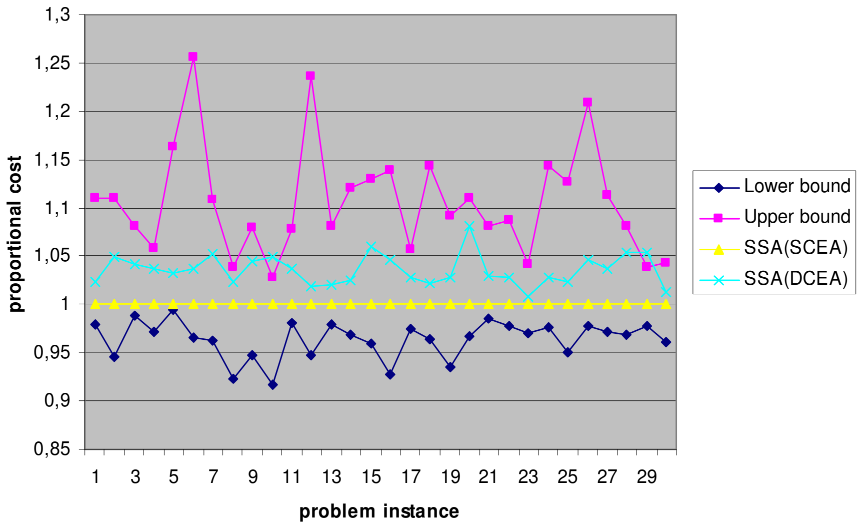


Figure 7. Comparing the proposed genetic algorithm (SSA) against the lower and upper bounds for problem instances of 12 jobs and 3 tools. The lower bound, upper bound and SSA(DCEA) costs are shown proportional to the SSA(SCEA) cost. The number of iterations for SSA is 10000, the size of the population is 200, the number of tournament rounds is taken from U(0,2) and mutation probability is 0.01. The tool lifetimes are taken from uniform distribution U(10,90), job processing times from U(1,10), the number of tools per job is 1 and the capacity of the tool magazine is 2. The upper bound uses the maximum lifetimes and the lower bound the minimum lifetimes. Two upper bound instances where not solved by the ILOG within time limit of one hour. The paired t-test shows that the lower bound and SSA results are from the same distribution with very high confidence (99%). The running times for SSA were less than one minute for all instances.

5.4 Approximating the benefit from stochastic information

The final test run measures the benefit from using stochastic information in the cost evaluation by comparing the results of DCEA and SCEA. Figure 8 shows that the average benefit is 7.8 %. In stochastic programming this benefit is called the value of the stochastic solution (VSSI), see Birge and Louveaux (1997). Based on the numbers presented in Birge and Louveaux (1997) for VSSI our 7.8 % can be considered as a good result.

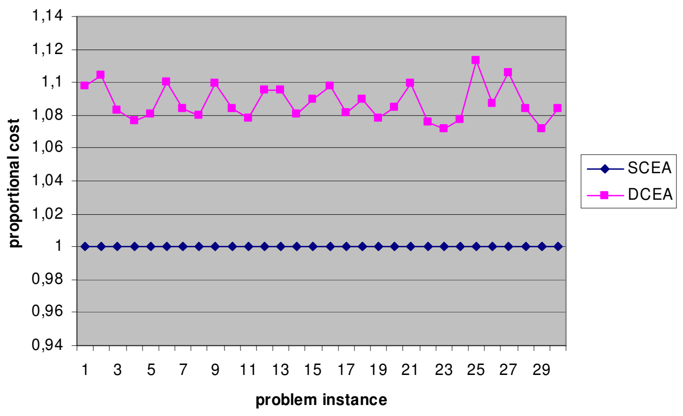


Figure 8. Comparing the cost evaluation algorithms for small problems. The values are based on the average 1000 random orders and they are calculated in proportion to the SCEA cost. The number of approximation rounds is 200 and the *u*-value is 0.0.

6. Conclusions

Stochastic scheduling and tool management was considered in this work. We gave a mathematical definition for the job scheduling with stochastic tool lifetimes (JSSTL) –problem. The complexity of the problem prohibited us from using standard methods, like scenario and analytical approaches for solving this problem. The complexity was due to the recursive nature of the problem which requires, in a case of tool break down, to return to the earlier stages of the processing of the jobs.

After defining the problem we considered lower and upper bounds for the JSSTL-problem. Two models LB_{max} and LB_{inf} for calculating the lower bounds were considered. The model presented in the appendix allows us to calculate a lower bound if we assume that all tools have some finite maximum lifetime. The model LB_{inf} assumes infinite lifetimes for all tools and is more general. This gives looser lower bounds but it also allows us to calculate the lower bound for problem instances without restricting the probability distributions. Another advantage is that LB_{inf} is somewhat easier to solve exactly because the number of discrete variables is considerably lower than in LB_{max}. The lower bounds can be calculated in restricted cases. The restriction is that the probability of a tool breakdown must be 0 for lifetimes smaller than any single job requires in it's processing, model LB_{min} is based on this assumption.

We presented two cost evaluation algorithms, which both rely on sampling. The processing cost is evaluated by calculating the average of several hundreds of samples. The DCEA algorithm does not take advantage of the stochastic information related to the problem because it uses expected lifetimes in the removal and reload decisions. The SCEA algorithm uses knowledge about the tool probability distributions. In the removal rules, one can adjust the used lifetimes through the *u*-value, which makes it possible to control the risk of a tool break down during the job processing. The algorithm uses also an approximate of the *EC*-function to evaluate the expected processing time of the next job with a certain tool magazine state. This method helps us to make better tool reload decisions. The genetic algorithm (SSA) is basically the same as GAPSM which was presented by Hirvikorpi et al. (2003). The major difference is that the search is guided either the by DCEA or SCEA.

The empirical results indicate that the SSA is able to solve the JSSTL-problem near optimally for small problem instances. The running times of SSA were acceptable for practical applications. The difference between DCEA and SCEA was statistically significant. On the average SCEA gave 7.8 % lower average cost than DCEA thus proving that using stochastic information in solving the JSSTL-problem is beneficial. The sampling method used in the approximation of the expected cost proved to be accurate enough when using reasonable number of samples.

The mathematical formulation of JSSTL is rather complex and it cannot be solved with the existing optimization packages. Simplifying the problem to a form which enables one to use traditional (i.e. analytical or scenario) approaches requires more work. The cost function also includes only the time cost, material costs incurring from discarded jobs are not included to the model. Taking these into account requires the use of multi-objective optimization which complicates the situation even further. Another, quite straightforward possibility, is to assign a monetary cost for the time used, too, and then minimize the total cost incurred.

References

Adiri I. J., Bruno E., Frostig A. H. G. and Rinnooy K., Single machine flow-time scheduling with a single breakdown, Acta Informatica, 26, 679-696, 1989.

Akturk M. S., Ghosh J. B. and Gunes E. D., Scheduling With Tool Changes to Minimize Total Completion Time: A Study of Heuristics and Their Performance, Naval Research Logistics, 50, 15-30, 2002.

Birge J. R. and Louveaux F., Introduction to Stochastic Programming, Springer Verlag, 1997.

Crama Y., Kolen A. W. J., Oerlemans A. G. and Spieksma F. C. R, Minimizing the Number of Tool Switches on a Flexible Machine, The International Journal of Flexible Manufacturing Systems, 6, 33-53, 1994.

Gray E., Seidmann A. and Stecke K. E., A Synthesis of Decision Models for Tool Management in Automated Manufacturing, Management Science, 39, 549-567, 1993.

Hirvikorpi M., Knuutila T. and Nevalainen O., Job ordering and management of wearing tools in Flexible manufacturing, Submitted for publication, 2003.

Knuutila T., Puranen, Johnsson M. and Nevalainen O., Three Perspectives for Solving the Job Grouping Problem, International Journal of Production Research, 39(18), 4261-4280, 2001.

Lee C. Y. and Liman S. D., Single machine flow-time scheduling with scheduled maintenance, Acta Informatica, 29, 375-382, 1992.

Lee C. Y., Machine scheduling with an availability constraint, Journal of Global Optimization, 9, 395-416, 1996.

Lee C. Y., Minimizing the makespan in two-machine flowshop scheduling problem with an availability constraint, Operations Research Letters, 20, 129-139, 1997.

Qi X., Chen T. and Tu F., Scheduling the maintenance on a single machine, Journal of the Operational Research Society, 50, 1071-1078, 1999.

Reeves C. R., Modern Heuristic Techniques for Combinatorial Problems, McGraw-Hill International, 1995.

Storer R. H., Wu D. S. and Vaccari R., New search spaces for sequencing problems with application to job shop scheduling, Management Science, 29, 273-288, 1983.

Tang C. S. and Denardo E. V., Models Arising from a Flexible Manufacturing Machine, Operations Research, 36(5), 767-84,1988.

Appendix A. Integer Formulation of the SWT-problem

The SWT-problem can be stated as follows, see Hirvikorpi et al. (2003). A set of jobs to be processed in a minimal average time with a single machine is given. A job requires a set of tools which must be placed in the magazine of the assembly machine before processing of the job begins. The tool set used by a job is a subset of of all tools and these sets are more or less similar to each others so that even identical sets may occure. The capacity of the tool magazine is limited and we assume that it is less than the joint capacity needed by all tools of the jobs. This problem is further complicated by the facts that each tool has a limited life-time (i.e. the time of proper functioning, when used), tool changes can only be done in between of jobs and each tool change takes certain amount of time to perform. It is supposed that the life-time of a tool is deterministic but it may be different for different tools. Further, the demand for tool changes between jobs means that the processing of a job can not be interrupted after it has been started.

Problem instance parameters (i=1,..., $|T|$, j,t=1,...,$|J|$):

c_i	switch time of tool i,
l_i	life-time of tool i,
p_j	processing time of job $j\in J$,
$p_{i,j}$	processing time of job $j\in J$ for tool $i\in T$,
T_j	the set of tools required by job $j\in J$, $T_j \subseteq T$,
$h_{i,j}$	1 if job j requires tool i, 0 otherwise,
K	capacity of the tool magazine as counted in the number of tools.

The decision variables are:

$x_{j,t}$	1 if job j is processed at time t, 0 otherwise,
$b_{i,t}$	remaining life-time of the tool i at time t, before processing t^{th} job,
$a_{i,t}$	remaining life-time of the tool i at time t, after processing t^{th} job,
$z_{i,t}$	1 if tool i is in the magazine at the time t, 0 otherwise,
$s_{i,t}$	1 if tool i is removed or added at the time t, 0 otherwise,
$r_{i,t}$	1 if tool i is reloaded at time t, 0 otherwise.

The SWT-problem can then be stated as:

minimize

$$\sum_{t=1..T} (P-t+1)(\sum_{j=1,...,P} x_{j,t}\, p_j + \sum_{i=1,...,T} (s_{i,t} + 2\, r_{i,t})\, c_i) \qquad (11)$$

subject to

$$a_{i,t} = b_{i,t} - \sum_{j=1..P} x_{jt}\, p_{j,t} \qquad (12)$$

$$\sum_{j=1,...,P} x_{j,t} = 1 \qquad (13)$$

$$\sum_{t=1,...,P} x_{j,t} = 1 \qquad (14)$$

$$z_{i,t} \geq \sum_{j=1..P} x_{j,t}\, h_{i,j} \qquad (15)$$

$$s_{i,t} \geq z_{i,t} - z_{i,t-1} \qquad (16)$$

$$s_{i,t} \geq z_{i,t-1} - z_{i,t} \qquad (17)$$

$$s_{i,t} \leq (1-z_{i,t})+(1-z_{i,t-1}) \qquad (18)$$

$$r_{i,t} \geq (b_{i,t} - a_{i,t-1}) / l_i - s_{i,t} \qquad (19)$$

$$\sum_{i=1,...,T} z_{i,t} \leq K \qquad (20)$$

$z_{i,t}\, l_i \geq a_{i,t},\ b_{i,t}$ (21)

$x_{j,t},\ z_{i,t},\ a_{i,t},\ b_{i,t} \in \{0,1\}$ (22)

$0 \leq a_{i,t},\ b_{i,t}$ (23)

$z_{i,0} = a_{i,0} = 0$ (24).

Constraint (12) binds the transition variables so that tool wearing actually occurs. The next two constraints guarantee that every job is processed exactly once. Constraint (16) says that all the tools required during each time period must be in the magazine. The following two constraints guarantee that whenever a tool is added or removed its decision variable *s* is set to one 1. Constraint (18) prevents the addition of a tool when the reload of that tool is needed. This is because the tool addition is cheaper in the objective function and we must prevent the adding when the tool is already in the magazine. Tool reload occurs when tool removal or addition is not made, but the life-time of the tool increases from the previous time period (19). Constraint (20) says that the given capacity must not be exceeded. If tool type is not in the magazine its life-time must be 0, this is formalized with constraint (21). The constraints (22) and (23) set the domains for the decision variables and constraint (24) says that we start with an empty tool magazine.

ISBN 951-29-2789-6
ISSN 1239-1883