Static and dynamic job-shop scheduling using rolling-horizon approaches and the Shifting Bottleneck Procedure

By

Ahmed S. Ghoniem

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
In partial fulfillment of the requirements for the degree of

Master of Science
In
Industrial and Systems Engineering
within the
Joint graduate program with
Ecole des Mines de Nantes - France

Dr. Subhash C. Sarin, Chairman – co-advisor
Dr. Stéphane Dauzère-Pérès, Co-advisor EMNantes
Dr. Hanif D. Sherali, Committee Member
Dr. Robert C. Williges, Program Chairman

September 2002
Nantes, France

Keywords: job-shop scheduling, rolling horizon approach, decomposition methods, shifting bottleneck procedure, static/dynamic scheduling.

Static and dynamic job-shop scheduling using rolling-horizon approaches and the Shifting Bottleneck Procedure

by Ahmed Ghoniem

Abstract

Over the last decade, the semiconductor industry has witnessed a steady increase in its complexity based on improvements in manufacturing processes and equipment. Progress in the technology used is no longer the key to success, however. In fact, the semiconductor technology has reached such a high level of complexity that improvements appear at a slow pace. Moreover, the diffusion of technology among competitors shows that traditional approaches based on technological advances and innovations are not sufficient to remain competitive.

A recent crisis in the semiconductor field in the summer 2001 made it even clearer that optimizing the operational control of semiconductor wafer fabrication facilities is a vital key to success. Operating research-oriented studies have been carried out to this end for the last 5 years. None of them, however, suggest a comprehensive model and solution to the operational control problem of a semiconductor manufacturing facility.

Two main approaches, namely mathematical programming and dispatching rules, have been explored in the literature so far, either partially or entirely dealing with this problem. Adapting the Shifting Bottleneck (SB) procedure is a third approach that has motivated many studies.

Most research focuses on optimizing a certain objective function under idealized conditions and thus does not take into consideration system disruptions such as machine breakdown. While many papers address the adaptations of the SB procedure, the problem of re-scheduling jobs dynamically to take disruptions and local disturbances (machines breakdown, maintenance...) into consideration shows interesting perspectives for research. Dealing with local disturbances in a production environment and analyzing their impact on

scheduling policies is a complex issue. It becomes even more complex in the semiconductor industry because of the numerous inherent constraints to take into account. The problem that is addressed in this thesis consists of studying dynamic scheduling in a job-shop environment where local disturbances occur. This research focuses on scheduling a large job shop and developing re-scheduling policies when local disturbances occur. The re-scheduling can be applied to the whole production horizon considered in the instance, or applied to a restricted period T that becomes a decision variable of the problem. The length of the restricted horizon T of re-scheduling can influence significantly the overall results. Its impact on the general performance is studied. Future extensions can be made to include constraints that arise in the semiconductors industry, such as the presence of parallel and batching machines, reentrant flows and the lot dedication problem.

The theoretical results developed through this research will be applied to data sets to study their efficiency. We hope this methodology will bring useful insights to dealing effectively with local disturbances in production environments.

Dedication

I would like to dedicate this thesis to my parents.

Acknowledgments

In the first place, I would like to express my gratitude towards Dr. Stéphane Dauzère-Pérès and Dr. Subhash Sarin for giving me the opportunity of working under their guidance. Their support and trust helped me in developing my own ideas and implementing them successfully. Dr. Stéphane Dauzère-Pérès helped me in carrying out my research for the Master's Thesis under ideal conditions in the Department of Automatic Control and Industrial Engineering at the Ecole des Mines de Nantes (France).

I would like to thank heartily Dr. Hanif Sherali who kindly accepted to serve in my committee. I appreciate much the support he gave me during my stay in Blacksburg (Spring 2001) and after I came back to France.

From February 2002 to September 2002, Fabrice Delmotte was my office mate in the Department of Automatic Control and Industrial Engineering at the Ecole des Mines de Nantes (France). I appreciated sharing the office with him and the friendly discussions we had together.

In addition, I would like to thank all the professors who contributed to the creation of this dual MS degree between Virginia Tech and the Ecole des Mines de Nantes. I am glad to be a pioneer of this promising program.

Finally, all my gratitude goes to my parents for what they did and do for our family and me.

Table of Contents

Table of Figures

1. Background

A. Introduction to the Problem

1. The Semiconductor Industry

The semiconductor industry has witnessed a steady increase in its complexity based on improvements in manufacturing processes and equipment over the last decade. Wafer fabrication is the most expensive and complex phase in the semiconductor industry. During this phase a wafer of silicon (or gallium) goes through 200 to 300 process steps in a clean room environment. Layers and patterns that convey electronic properties to the wafer are built up over weeks of time. Since the manufacturing process involves very sophisticated, and thus, capital-intensive tools, finding optimal production, and scheduling policies and service contention is crucial.

Semiconductor wafer fabrication is a multi-stage process divided mainly into the following sub-processes: photolithography, diffusion, metalization, doping and etching. Each sub-process involves sophisticated tools arranged in a manner similar to that in a job shop. Finding an optimal schedule for the different tools is a very difficult problem. Besides the large number of process steps, the complexity of the problem is due to the large number of lots and tools in a fab. There are also specific constraints inherent to semiconductor manufacturing. In fact, the scheduling problem on hand includes various types of tools (single, parallel and batching machines), re-entrant flows of products and some other random disturbances.

Traditional approaches consider technological advances in both processes and equipment as the key to success and the right way to remain competitive. This is no longer a valid approach, since these advances are shared between competitors very rapidly and none of them can have them exclusively. On the other hand, improvements in the scheduling and

production policies can lead to dramatic differences in the performance of the semiconductor fabrication.

Optimizing the scheduling policies of a semiconductor fab can shorten production cycle time significantly. It will thus lead to a better production flexibility and manufacturing cost reduction.

2. Specific constraints

The famous shifting bottleneck (SB) procedure (cf. Chapter 3) is originally proposed by Adams, Balas and Zawack (1988). It is a heuristic for the general job-shop problem and uses a disjunctive graph representation to capture interactions between single machines. Due to constraints specific to semiconductor manufacturing, this heuristic cannot be applied without being adapted to the problem on hand.

The first issue relates to *the dynamic behavior* inherent to any production system, including machines breakdowns and the arrival of new jobs in the system. A third factor, namely reentrant flows, makes this issue even more complex in a semiconductor fabrication facility.

The second issue relates to the fact that lots waiting for being processed on a given tool can be at different stages of fabrication, that is, they could either be at the start of their route, at the end, or somewhere in between. If one of the above-listed classes dominates over the others, then it has the highest priority for scheduling.

In addition, the number of wafers to have in one lot depends on the transit time from one tool to another. If this transit time is negligible, then it would be optimal to reduce the number of wafers in a lot. If not, then, larger lot sizes are more beneficial.

The fourth issue relates to the presence of various types of machines: single machines or parallel machines, and also batching machines, i.e. machines capable of processing multiple lots at a time. Some machines can process only one lot at a time, and are referred to

as "Single Lot at a Time" processing machines. Other machines, however, like ovens or some types of steppers, can process more than one lot at the same time. These are called batching stations and must be taken into account as such. The problem is complicated further by the fact that some process steps are processed by parallel identical machines. For example, the photolithography department contains many identical steppers. As a consequence, the problem on hand requires solution of a parallel processor scheduling subproblem.

The fifth subproblem relates to equipment alignment and calibration issues. Due to quality concerns, some process steps have to return to the same piece of equipment that processed the lot in a previous critical step. This subproblem will be referred to as the "lot dedication problem".

Capturing all of these constraints in one heuristic is a very complex problem, one that has not been solved yet. The problem becomes even more difficult if re-scheduling steps to deal with breakdowns and disruptions are taken into consideration.

This research focuses on some aspects of the dynamic behaviour, namely scheduling jobs and re-scheduling them when local disturbances occur in the production system. This work should lead to future extensions where more constraints will be added to better represent the semiconductor environment.

3. Dynamic behaviour

The dynamic behaviour in a semiconductor fabrication facility is due to three main factors.

a. There are specific requirements for processing jobs in a semiconductor manufacturing facility. First, jobs have to be processed by the same tool (or at least the same type of tool), at different stages of their fabrication. This creates re-entrant flows (also referred to by re-circulation) within the production system. Jobs, therefore, do not progress linearly in the manufacturing system.

b. The dynamic behaviour is also due to new lots introduced into the production system. In fact, the shifting bottleneck procedure deals with “static environments”, where an initial set of jobs goes through a job shop, with no new arrivals in the system, and no re-entrant flows of products. In an industrial environment, the set of lots in queue at a particular station must be updated regularly because of dynamic arrivals, thus the schedule given by the adapted shifting bottleneck procedure will be modified accordingly.

c. Another important factor causing a dynamic behaviour within the production environment consists of local disturbances. Local disturbances include machine breakdowns and other events resulting in disruption of a piece of equipment. In fact, local disturbances through equipment can be due to a machine breakdown or to processing a “troublesome” lot of wafers. An illustration of troublesome lots can be given in the Photolithography Area. If some lots of wafers do not behave smoothly during the alignment step of a stepper, the stepper would not be able to align the wafer. This would result in an error (the machine is said to be in a Z-mode, which eventually requires the operator to intervene). When most wafers in a lot produce such an error, the processing time of this particular lot can increase dramatically thereby causing a disruption of the stepper.

B. Problem Definition

The problem that we address can be stated as follows: Given a large job shop and a number of jobs, determine how should the jobs be scheduled and how should they be re-scheduled when local disturbances occur, so that the makespan (Cmax) is minimized**?** The problem addressed in this research consists of studying the dynamic scheduling of jobs in a job shop where local disturbances (such as machine breakdowns) occur. Given a large job shop, the Jm||Cmax problem will be studied. Several approaches will be used: the classical SBP and Decomposition Heuristics. This will result in a primary schedule. Then, dynamic scheduling due to local disturbances will be addressed. This research will consider machine

breakdowns and events that can be modeled as a machine breakdown. The re-scheduling step can basically be applied to the whole production horizon or to a **restricted period T** which becomes a decision variable of the problem. The length of the restricted horizon T, considered in the re-scheduling phase, can influence the re-scheduling approach and thus impact the overall results significantly. Its impact on the general performance and the re-scheduling methods used are studied. Classical job shops with no flexible machines are also considered. Future extensions can be made to include more constraints that arise in the semiconductor industry, such as the presence of parallel and batching machines, reentrant flows and the lot dedication problem.

1. Preliminary note

The job shop problem can be described as follows: a set of *n* jobs has to be processed on a set of *m* machines. Each job has a release date *rj* and a specific routing through the different machines. Operation (i, j) refers to job j being processed on machine i. *pij* is the processing time of job j on machine i.

Let *Cj* denote the completion time of job j on the last machine in its routing. The makespan is defined as *Cmax* = max(*Cj*). The *Jm*| *rj* | *C*max, where Cmax is minimized, has motivated a significant amount of research. Wein and Chevalier [45] underline the fact that minimizing Cmax is equivalent to minimizing the cycle time. By Little's law, this implies maximizing the throughput, thus minimizing costs. Two main types of approaches have been explored to solve this problem: exact methods and heuristics.

Among the heuristics developed for job shop scheduling, the SBP proposed by Adams et al. [1] is one of the most famous ones. This procedure is explained in chapter 3 and will be used in this research. Although the results for small-size problems are satisfactory, more research need to be carried out to better deal with middle- and large-size problems. A

significant amount of research has been devoted to extend or adapt the SB procedure, as detailed in chapter 2.

2. Phase 1: Scheduling phase

Disjunctive graphs are useful tools to represent job shops and to show the interaction between jobs to be scheduled on the same machine. Figure 1 shows a disjunctive graph. Chapter 3 presents disjunctive graphs in more details. This first issue addressed in this research consists of depicting a large job shop with a disjunctive graph representation and exploring various scheduling approaches to minimize Cmax.

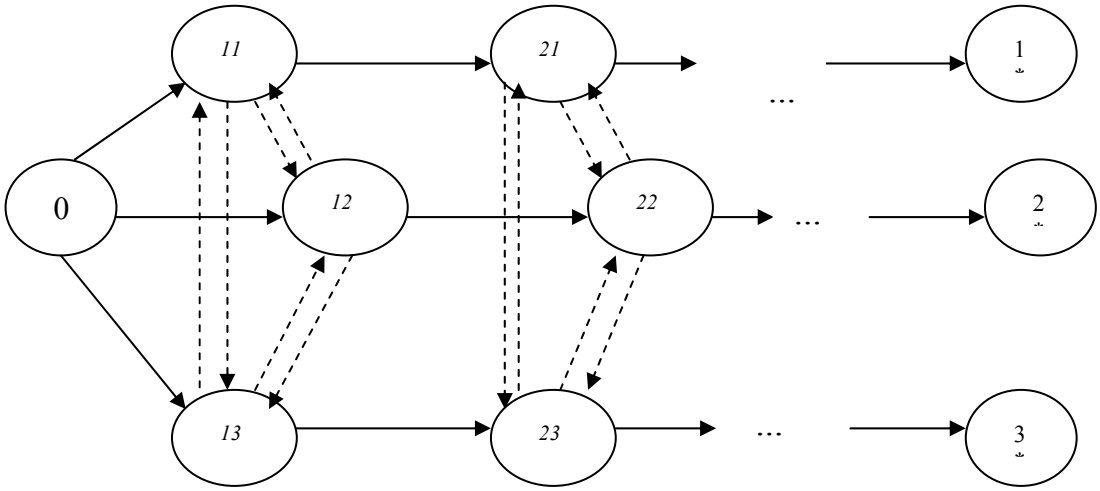

Figure 1 - A Disjunctive Graph

Two main approaches are studied. First, the classical SB procedure will be applied. Then, a rolling horizon heuristic based on decomposition methods will be developed. This method divides a given instance into a number of sub-problems, each corresponding to a time window of the overall schedule, and all of which are solved using a shifting bottleneck heuristic or other scheduling approaches. Figure 2 shows a disjunctive graph divided into several sub-windows. Both techniques will generate a schedule to the global problem. It is

interesting to compare the results of both approaches and, in phase 2, the impact of disruptions and local disturbances on schedules generated in phase 1 will be studied.

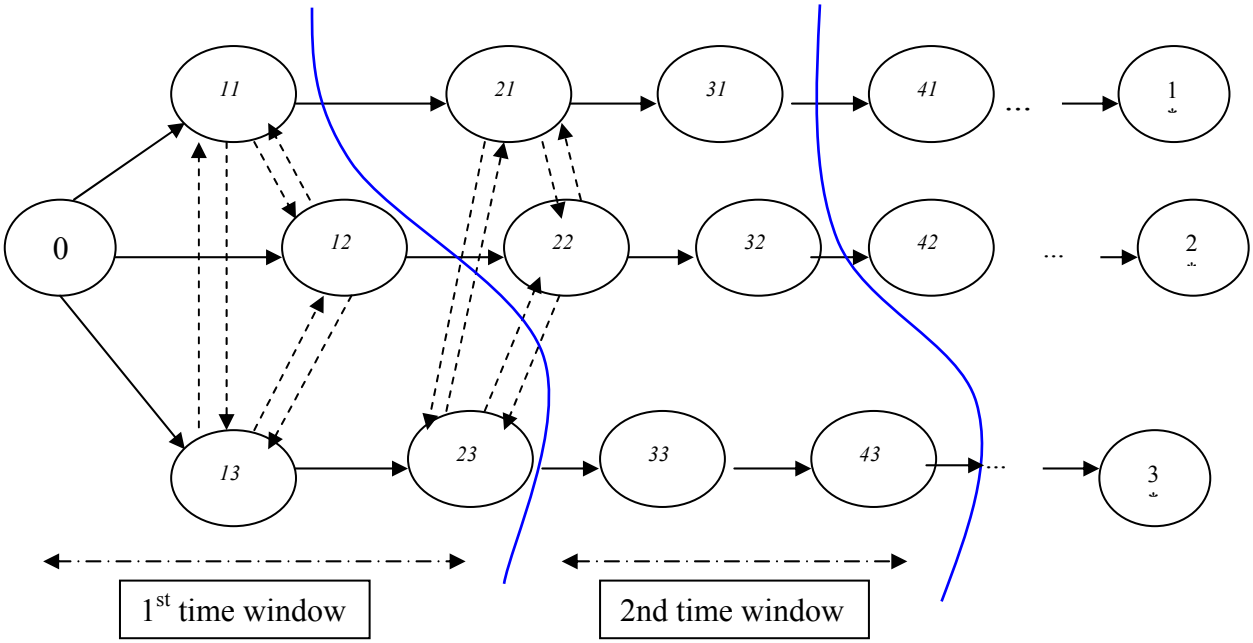

Figure 2 - Disjunctive graph with time windows

3. Phase 2: Rescheduling phase

The disjunctive graph depicted in figure 3 represents a 10x10 job shop (10 machines and 10 jobs) divided into several sub-windows. It also highlights the nodes affected by a breakdown of machine 4. This problem can be solved with the SBP or with a RH heuristic. Both methods generate, in phase 1, a schedule for the whole problem. With a RH heuristic, the instance is divided in subproblems, corresponding to time windows of the overall problem, as depicted in figure 3. Each time window is limited by (blue) plain borders. Between t = t0 and t1, the schedule generated in phase 1 is applied, and several jobs are processed. At t = t1, a machine breakdown occurs on machine 4. At this point a "re-scheduling phase" starts to deal with this disruption.

Decisions taken for nodes on the left side of the (red) dashed border (i.e. jobs processed before the breakdown occurs) are fixed. However, nodes on the right side of this (red) dashed border can be re-scheduled to integrate the fact that one of the machines is not available for a certain amount of time.

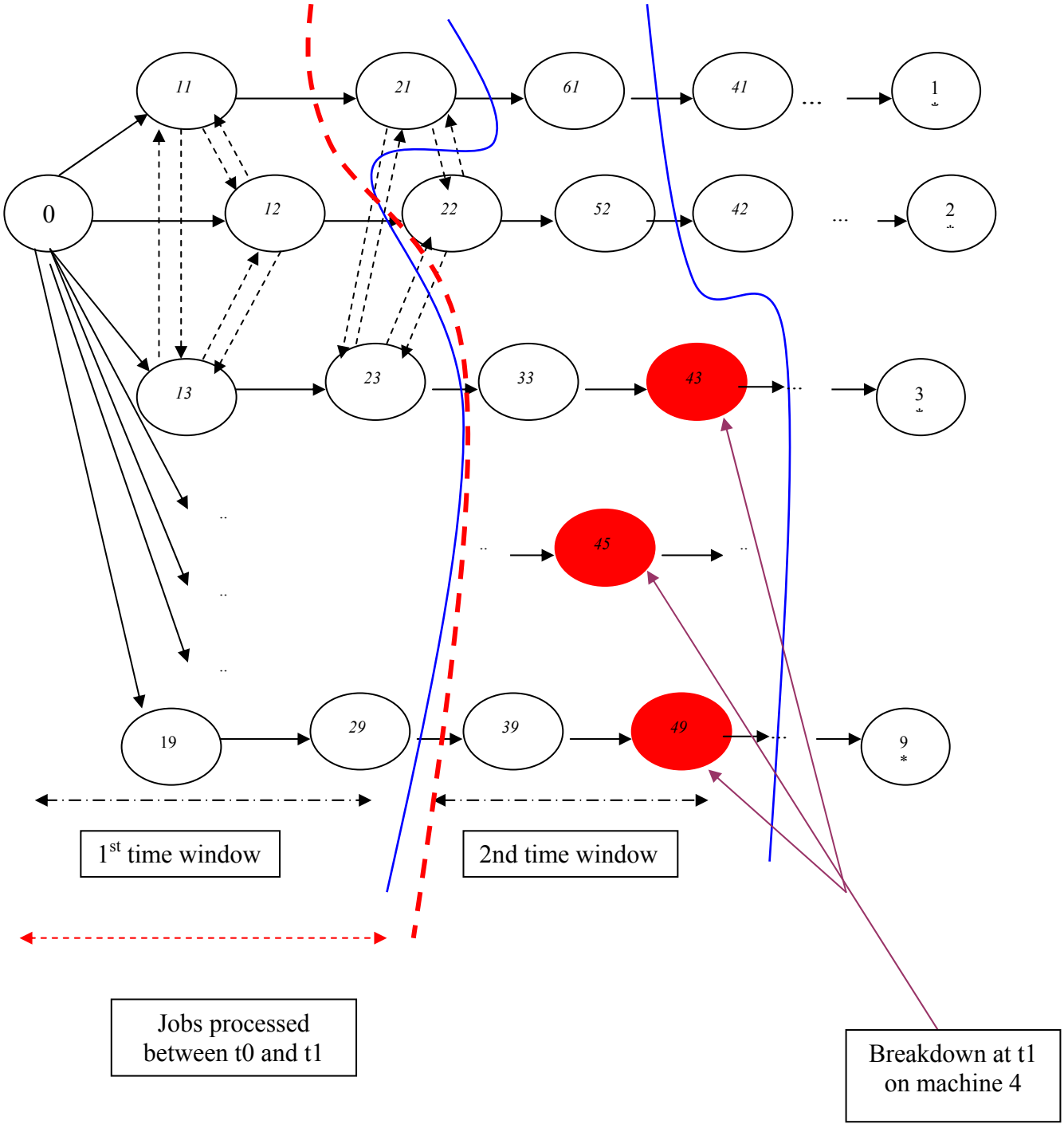

Figure 3 - Disjunctive Graph with time windows in the event of a system disruption

Instead of re-scheduling jobs over the horizon T2, it can be interesting to re-optimize the problem on a restricted re-scheduling window time, T1 (as depicted in Figure 4). T1 is a decision variable of the problem that is influenced by many factors: the length of the breakdown, the number of jobs that have to be processed on the unavailable machine etc.

Assuming that portion A of the graph (left side of the rescheduling time window, T1) and portion C (right side of T1) are fixed according to the schedule generated in phase 1, phase 2 aims to study the length of T1 and its consequences on the re-optimization approach and the overall results.

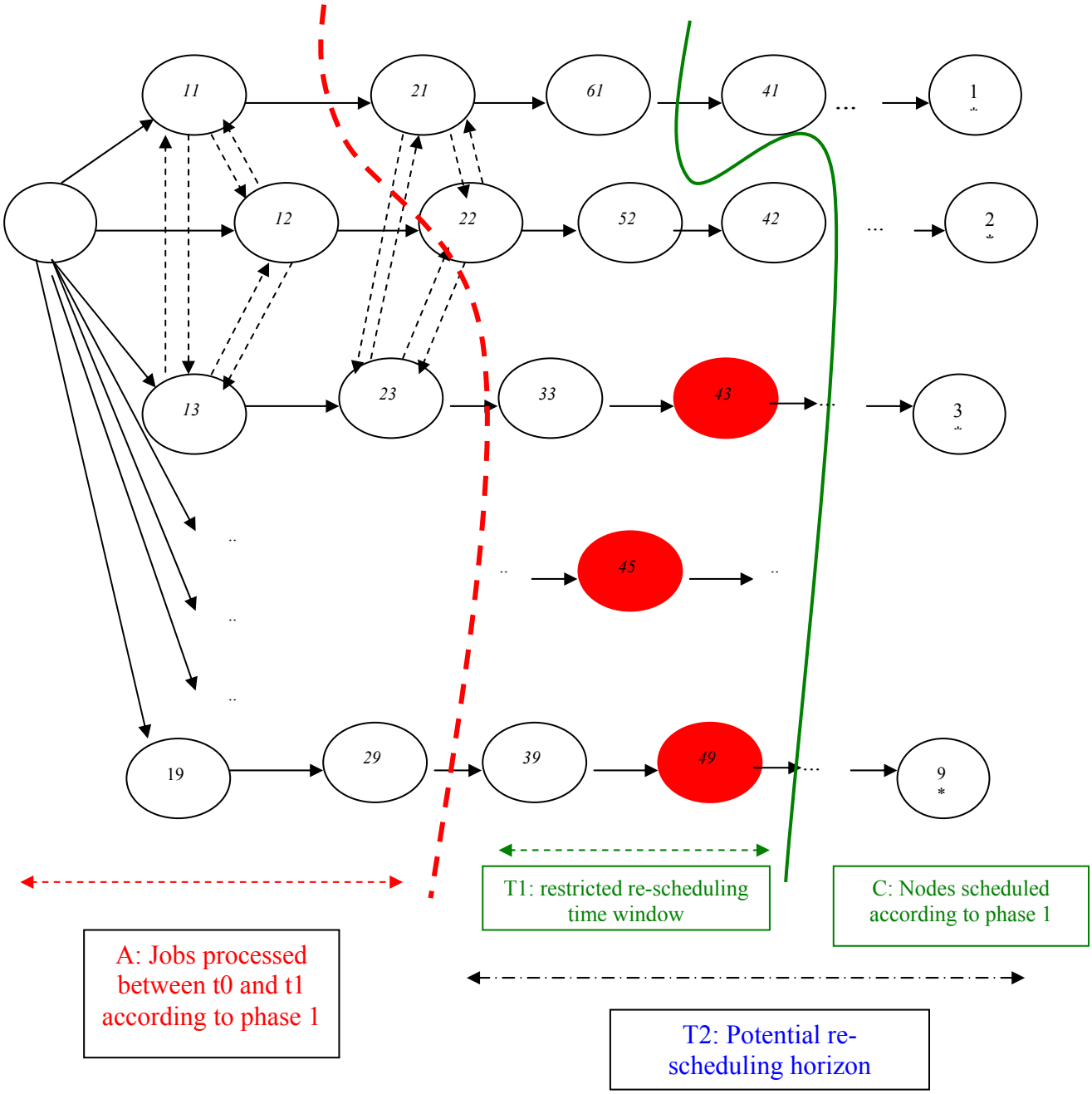

Figure 4 - Disjunctive Graph and the re-scheduling phase

C. Research Goals

1. Research Questions

- In a given large job shop, how should the jobs be scheduled and how should they be re-scheduled when local disturbances occur, so that the makespan Cmax is minimized?
- Can a rolling horizon heuristic outperform the classical SBP in minimizing Cmax for a large job shop?
- How should the length of a time window be chosen to divide a large horizon while using a RH heuristic?
- How should the length of the re-scheduling time window be determined? What parameters influence this decision?
- What re-scheduling techniques should be applied?

2. Problem Hypotheses

Instances of the job shop problem (10x10, 5x10, 5x15, 5x20) are considered. Three main parameters can influence the decision variables. The first parameter is the length, L, of the breakdown. Several cases with different values of this parameter are studied. However, in each case, it is assumed that the length of the breakdown is known *as soon as* it occurs. The second parameter is the machine on which the breakdown occurs. The third parameter is the time t at which the breakdown occurs. This parameter has a direct consequence on the number of nodes that are fixed and indirectly reflects on the total number of nodes present in the potential re-scheduling horizon. The different simulations tested in this research consider various values of these three parameters.

Concerning the operation which is processed when the breakdown occurs, two scenarios can be considered: re-scheduling with preemption or without preemption.

3. Research Objectives

The primary objective of this research is to study, model, analyze and implement a heuristic to minimize the makespan in a large job shop in the event of a system disruption, such as machine breakdown.

The aim of this research is to achieve the following:

1. Study the scheduling of a job shop through two techniques: the SBP and a RH approach.
2. Study the performance of reactive scheduling in the event of a system disruption.
3. Develop different approaches to deal with local disturbances by simulating several cases where the parameters defined in the "research hypothesis" vary.

4. Significance of Proposed Research

Most research reported in the literature focuses on optimizing a certain objective function under idealized conditions and thus do not take into consideration system disruptions such as a machine breakdown. This research considers in phase 1, the optimization of a large job shop under idealized conditions to minimize the makespan, then, in phase 2, it deals with reactive scheduling in the event of a machine breakdown or events that can be modeled as such.

This work can lead to applications in production environments that can be modeled as a job shop. It develops scheduling approaches for the case where no disruptions occur. It also suggests and analyzes re-scheduling approach to deal with local disturbances when they do occur.

This research will complement the research devoted to optimizing a client-oriented objective function (such as the total tardiness) with RH approaches. In fact, Singer [20] considered the problem of minimizing the total weighted tardiness in a job shop which orients the optimization to the customer's satisfaction rather than to cost reduction. On the other

hand, minimizing Cmax is equivalent to minimizing the cycle time, which implies maximizing the throughput rate and minimizing costs.

Furthermore, future extensions can be made to include more constraints that arise in the semiconductor industry, such as the presence of parallel and batching machines, reentrant flows and lot dedication.

2. Literature Review

A. Semiconductor Manufacturing

Most research carried out on production control issues in a semiconductor manufacturing facility discusses or explores two approaches: dispatching rules and mathematical programming [31].

The use of Dispatching Rules (DR) constitutes the most common approach developed and used so far to schedule a semiconductor facility [46]. Studies on their efficiency [31,13] reveal that their performance is limited since they are not flexible enough to the changes encountered in the system conditions. Various papers address the control of workload at the bottleneck workstation [30, 24, 29].

Minimizing the WIP has been addressed with linear program structures [36]. Research has been carried out to combine LP and non-LP in hierarchical structures [4]. Combined approaches such as DR and integer programming [16], or LP and simulation [20 & 29] have been addressed. Queuing Network models have not been applied to the scheduling problem. They have been used, however, to study various parameters in production systems such as fab layout, setup time, product mix etc. [34, 16, 10]

B. Shifting Bottleneck Procedure

Among the heuristics developed for job shops, one of the most successful approaches is the Shifting Bottleneck proposed by Adams et al. [1] and is explained in Chapter 3. Dauzere-Peres and Lasserre [11], Applegate and Cook [3] and Balas et al. [5] also enhanced this method. Van Laarhoven et al. [44], Taillard [42], Barnes and Chambers [6] and Nowicki and Smutnicki [36] presented some developments on local search methods. Ramudhin and Marier [38] generalize the SBP to open-shop problems.

Some papers have focused on quality-oriented (due-date oriented) functions, whereas optimizing the makespan aims to maximizing the throughput and thus minimizing costs. Such research focuses on the effectiveness rather than the efficiency of the job shop in meeting clients' requests. This objective is gaining in priority since companies compete not only in their prices but also in their punctuality and their meeting the agreed upon due dates.

Assume that each job *j* has a due date *dj* and a weight *wj* , and define L*j* = *Cj* - *dj* as the lateness of job *j* and T*j* = max(L*j* , 0) as the tardiness of job *j*. Ovacik and Uzsoy [37] have modified the Shifting Bottleneck heuristic to solve the *Jm*| *rj* , *sij*| Lmax problem, where *sij* denotes the presence of sequence-dependent setup times.

Some papers address the problem of minimizing the total weighted tardiness in a job shop, denoted by *Jm* | *rj*| Σ *wj*T*j*. This problem is strongly NP-Hard since it is a generalization of the single-machine problem 1|| Σ *wj*T*j* , which Lenstra et al. [28] proved to be strongly NP-Hard. Pinedo and Singer develop a Shifting Bottleneck heuristic for minimizing the total weighted tardiness (SB-TWT) and Singer [39] presents a heuristic that decomposes the problems on a time window basis, solving each subproblem using a shifting bottleneck heuristic. He shows that the results for a due-date-related objective function are promising.

C. Reactive Scheduling

The majority of the published literature in the scheduling area deals with the task of schedule generation or predictive nature of the scheduling problems. But, reactive scheduling and control is also important for the successful implementation of scheduling systems. In what follows, we review the research papers that are related to reactive scheduling.

The first study in this area is due to Holloway and Nelson (1974) who implement a multi-pass procedure (as described later in Nelson et al., 1977) in a job shop by generating schedules periodically. They concluded that a periodic policy (scheduling/rescheduling periodically) is effective in dynamic job shop environments [18].

Later, Farn and Muhleman (1979) compared dispatching rules and optimum seeking algorithms for the static and dynamic single machine scheduling problems. Again, new schedules are generated periodically in a dynamic environment. Their results indicate that the best heuristic for a static problem is not necessarily the best for the corresponding dynamic problem [14].

In addition, Muhleman et al. (1982) analyze the periodic scheduling policy in a dynamic and stochastic job shop system. Their experiments indicate that more frequent revision is needed to obtain better scheduling performance. [33]

Church and Uzsoy (1992) consider periodic and event driven (periodic revision with additional considerations on tight due date jobs) rescheduling approaches in a single machine production system with dynamic job arrivals. The results indicate that the performance of periodic scheduling deteriorates as the length of rescheduling period increases and event driven method achieve a reasonably good performance. Later, Ovacik and Uzsoy (1994) propose several rolling horizon procedures in a single machine environment with sequence dependent set-up times [9].

Kiran et al. (1991) propose another rolling horizon type heuristic for manufacturing systems. The experiments with their model in a dynamic environment indicate that the proposed heuristic performs well for several tardiness related criteria. [23]

Yamamoto and Nof (1985) study a rescheduling policy in a static scheduling environment with random machine breakdowns. Rescheduling is triggered whenever a machine breakdown occurs. The results indicate that the proposed approach outperforms the fixed sequencing policy and dispatching rules. [49]

Also, Nof and Grant (1991) develop a scheduling/rescheduling system and analyze the effects of process time variation, machine breakdown and unexpected new job arrival in a manufacturing cell. In their scheduling system, monitoring is performed periodically and

either rerouting to alternative machines or order splitting policies are activated in response to unexpected disruptions. [35]

Bean et al. (1991) consider the rescheduling of the shop with multiple resources when unexpected events prevent the use of a preplanned schedule. The authors reschedule to match-up with the preschedule at some point in the future whenever a machine breakdown occurs. The match-up approach is compared with the no response policy and several dispatching rules. The results of the test problems indicate that the proposed system is more advantageous. [7]

Later, Akturk and Gorgulu (1998) apply this approach to the modified flow shop. The results indicate that the match-up approach is effective in terms of schedule quality, computation times, and schedule stability. [2]

Simulation based approaches are also widely reported in the scheduling literature. various control policies are tested by using simulation.

For example, Wu and Wysk (1988, 1989) propose a multi-pass scheduling algorithm that utilize simulation to make scheduling decisions in an FMS. Specifically, the multi-pass scheduling system simulates the system for each alternative rule by using the current shop status information and selects the best rule to implement. The results show that the multi-pass approach is considerably better than using a single rule for the entire horizon [47 & 48].

Jain and Foley (1987) use the simulation methodology to investigate the effects of the machine breakdowns in an FMS. Their experiments indicate that rerouting is always a better policy. [21]

Matsuura et al. (1993) study the problem of selection between sequencing and dispatching as a rescheduling approach in a job shop environment involving machine breakdowns, specification changes, and rush jobs. The authors propose a method that switches from

sequencing to dispatching when an unexpected event occurs. Their results show that this combined approach performs very well. [32]

In another study, Kim and Kim (1994) develop a simulation-based real time scheduling methodology for an FMS. In this system, there are two major components: a simulation module and a real time control system. The simulation module evaluates various dispatching rules and select the best one for a specified criterion. The real time control module monitors the shop floor and a new schedule is generated at the beginning of each period when there is a major disturbance in the system. [22]

Also Bengu (1994) develops a simulation-based scheduler that uses the up to date information about the status of the system and improves the performance of a scheduling rule Apparent Tardiness Cost (ATC) under dynamic and stochastic production environments. [8]

Kutanoglu and Sabuncuoglu (1994) compare four reactive scheduling policies under machine breakdowns. The policies (all rerouting, arrival rerouting, queue rerouting and no rerouting) are tested using a job shop simulation model. A material handling system (MHS) is also considered in the model. The results show that the all rerouting is preferred to reactive policy when the MHS is ignored [25]. In the later study, Kutanoglu and Sabuncuoglu (1998a,b) propose an iterative simulation based scheduling mechanism for dynamic manufacturing environments. The authors test the proposed method by using multi-pass rule selection algorithm and lead-time iteration algorithm in both deterministic and stochastic environments. The results indicate that the iterative improvement procedure improves the performance of the dispatching rules significantly. [26 & 27]

Later, Sabuncuoglu and Karabuk (1997) study the scheduling rescheduling problem in an FMS environment. The authors propose several reactive scheduling policies to cope with machine breakdowns and processing time variations. Their results indicate that it is not always beneficial to reschedule the operations in response to every unexpected event and the

periodic response with an appropriate period length can be quite effective in dealing with the interruptions. [40]

Hsieh et al (2001) investigated selections of dispatching rules at the occurrence of significant WIP holding and major machine failure. They exploited the speed of an (ordinal optimisation) OO-based simulation tool to select a good scheduling rule for the coming four weeks. Four prominent dispatching rules (FSVCT [1], FIFO, LDF [2] & OSA [3]) combined with the workload regulation release policy constituted a set of 256 rule options over a 4-week horizon, where the dispatching rule might change weekly. Their conclusion was that dispatching rule should be switched from the slack time-based FSVCT to the deviation-from-target-based LDF to handle unusual events. These observation justified that dispatching rules should be changed dynamically to handle machine failures and other significant disruptions. [19]

The reactive scheduling problems are also studied by using knowledge based and artificial intelligence (AI) techniques.

For example, Dutta (1990) develops a Knowledge Based (KB) methodology to perform real time production control in FMS environments. The proposed mechanism monitors the system and takes a corrective action whenever a disruption event occurs. The author considers machine failures, dynamic introduction of new jobs and dynamic increases in job priority as shop floor disruptions. The results show that the KB mechanism with such corrective actions renders effective and robust production control. [12]

---

[1] FSVCT: Choose the lot with the smallest (a_n + Cp – Di), where p is the index of product type, a_n is the release time of lot n, Cp is the mean cycle time, and Di is the estimate of the remaining cycle time from buffer i.

[2] LDF: Let the completion time of one wafer processing at a stage be a move. Choose a stage with the largest deviation of completed moves from the desired moves, where the desired number of moves of each product type at each stage is pre-specified. Then choose from the stage a lot that is released into the fab the earliest.

[3] OSA: Let Ni(t) be the WIP at time t at stage i, Ai is the average WIP at stage i. Choose a stage according to the following priorities :

- Priority I: stage i such that Ni(t) > Ai and Ni+1 (t) < Ai+1
- Priority II: stage i such that Ni(t) < Ai and Ni+1 (t) < Ai+1
- Priority III: stage i such that Ni(t) > Ai and Ni+1 (t) > Ai+1
- Priority II: stage i such that Ni(t) < Ai and Ni+1 (t) > Ai+1

Choose a lot with the same priority using FSVCT.

There are also other AI based studies in the literature. Among them, ISIS developed by Fox and Smith (1984) and OPIS proposed by Smith et al. (1990) are the most well-known systems. [15]

Other AI or KB systems are can be found in Szelke and Kerr (1994). [41]

There are other studies that investigate scheduling problem under certain stochastic events and variations. He et al. (1994) examine the effect of processing time variation (PV) on the dispatching rules and find that the relative performances of the rules remain the same under PV. [17]

Using decomposition methods for dynamic problems is an approach that has not been fully explored yet. Uzsoy and Perry considered the reactive scheduling of a Semiconductor Testing Facility using a workcenter-based decomposition method (inspired from the SBP):

- Step 1: Divide the job shop into workcenters. These could be single, parallel or batching machines.
- Step 2: Represent the job shop using a disjunctive graph.
- Step 3: Sequence each workcenter whose sequence has not been determined yet. Workcenters are ranked in order of criticality according to the objective function. The sequence of the most critical workcenter is fixed.
- Step 4: Use the disjunctive graph representation to capture the interactions between the workcenters already scheduled and those not yet scheduled.
- Step 5: Use the new information obtained in step 4 to re-sequence workcenters that have already been sequenced. If all workcenters have been scheduled, STOP. Else, return to step 3.

Their approach combined this workcenter-based decomposition method with an Event-driven Rescheduling approach. The motivation behind their research is the increasing need for companies to meet due dates so as to maintain high levels of on-line delivery. Preliminary

experiments showed that their rescheduling policies had performance comparable to myopic dispatching rules. [43]

3. Methodology

A. Disjunctive graph

Disjunctive graphs with single machines are briefly presented in this section. A disjunctive graph is made of a set of nodes and two sets of arcs, conjunctive and disjunctive. Its components are as follows:

- A set of n jobs, designated by N, to be scheduled.
- Every (i,j) node represents an operation where job j processed on machine i. We assume here that having job j processed on machine i is "one" operation.
- A set of conjunctive arcs designated by A. A conjunctive arc is a simple directed arc (→) used between (i,j) and (k,l), ((i,j) → (k,l)), if operation (i,j) must precede (k,l).
- A set of disjunctive arcs designated by B. A disjunctive arc consists of a pair of arcs having opposite orientations: ↔. Any real path though the whole graph contains at most one of these two arcs. A disjunctive arc between operations (k,j) and (k,l) means that both jobs j and l have to be processed at workcenter k with no precedence constraints.
- A cost associated with the arc emanating from any node (i,j) and designated by cij (= pij), where pij is the processing time of job j on machine i.
- A dummy source node (0) is added. It is linked with conjunctive arcs to all first operations of the jobs. The cost associated with each of these arcs is zero. In other words, for all j, $p_{0j} = 0$.
- A sink node (i*) represents the completion of job i. The last operation (m,j) of job j is linked to the (j*) node via a conjunctive arc. The cost associated

with this arc is p_{mj}. Instead of using a sink node for each job, one sink might be used for the whole problem.

Example

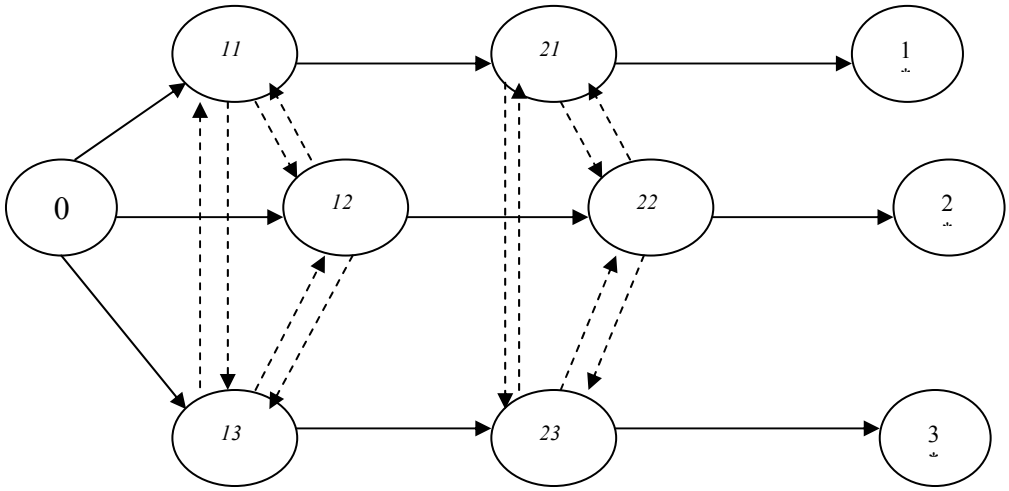

Figure 5 - A Disjunctive Graph

B. The Shifting Bottleneck Procedure

The shifting bottleneck (Adams, Balas and Zawack (1988)) is a very successful heuristic for J_m // C_{max}. Dauzere-Peres and Lasserre [11], Applegate and Cook [3] and Balas et al. [5] enhanced this method. Numerical research has shown that this procedure is very effective. In this research, Carlier's algorithm – a very efficient branch and bound procedure - is used to choose the machine causing the most severe disruption with the Shifting Bottleneck.

Let M denote the set of all m machines and Mo the set of machines that have been already scheduled. At each iteration, a set of disjunctive arcs is added to set Mo.

The different decisions which are taken at an iteration are as follows:

- Selection of a machine from M-Mo to be included in set Mo. The machine to be selected, according to Carlier's algorithm, is the one, from among unscheduled machines, that causes the severest disruption,. To determine this, the original directed graph is modified

by deleting all disjunctive arcs of the machines still to be scheduled (i.e. the machines in M-Mo) and keeping only the conjunctive arcs of the machines already sequenced. The resulting graph is called G'. The Graph G' has one or more critical paths that determine the corresponding makespan. For a given operation (i,j), where i belongs to the machines in M – Mo, the due date and the release date are deduced from the critical path in G'. For each of the machines in M-Mo, Carlier's algorithm is applied. After solving all these single machine problems, the machine for which Carlier's algorithm gives the largest value is chosen. This machine is the "bottleneck" among the machines in M - Mo and, thus, it is the one to be added next to Mo. Label this machine k.

- Fix the job sequence for machine k according to the optimal solution obtained with Carlier's algorithm. If the corresponding disjunctive arcs, which specify the sequence of operations on machine k, are inserted in graph G', then the makespan of the current partial schedule will increase.
- An optional re-optimization step in which all the machines in Mo are re-sequenced may be performed.

C. The rolling horizon approach

1. Overview

Solving large job shops with exact methods or heuristics such as SBP is time consuming. Rolling horizon approaches aim to divide a large instance into several subproblems (time windows) that are optimized independently. As every subproblem is optimized, the schedules corresponding to the different time windows are merged together to get the schedule for the initial problem.

To avoid the scheduling of a time window to interfere with the following time window, an overlapping degree can be defined. The degree of overlapping determines the number or the percentage of operations that are frozen or scheduled when a time window is scheduled. Let ω denote the size of the time window subproblem to be optimized, let Φ (≤ ω) denote the number of operations that will be fixed by the solution of the subproblem. In the example of figure 6., let ω = 3 and Φ = 2.

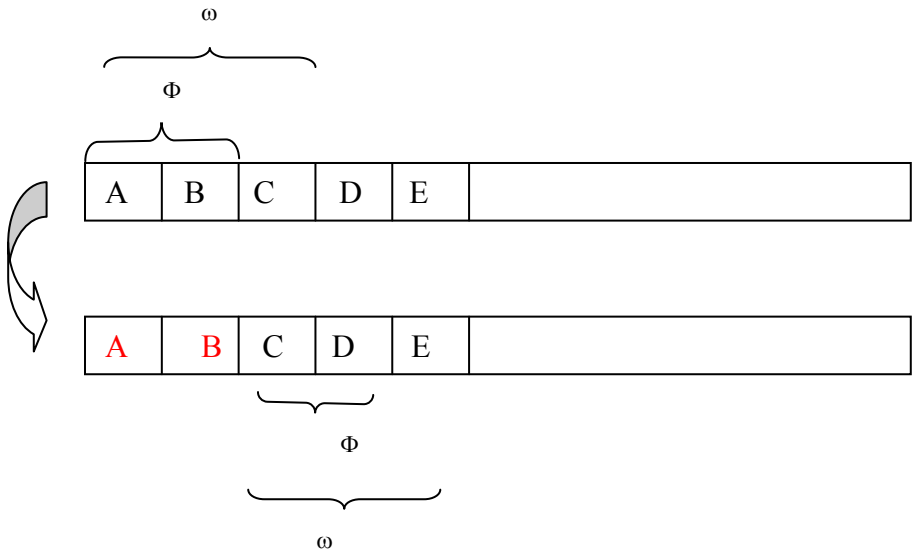

Figure 6 - Rolling Horizon Heuristic on a single machine

In general, three decisions are involved in this heuristic:

1. How big should a time window be?
2. What is the overlapping degree allowed between time windows ?
3. How is each time window optimized?

2. Optimization of each time window

Once the total instance is divided into several subproblems to be solved, each time window can be optimized. Several optimization techniques can be studied and applied (among others, the SBP and various dispatching rules).

If the subproblems are uncoupled, each one of them can be optimized independently from the others. If problems are interrelated, however, it is not possible to optimize a subproblem without integrating the optimization of the others. With a disjunctive graph approach, the external links between subparts of the graph are translated into due dates and release dates. If it is not possible to uncouple the different subproblems fully; it might, however, be possible to have some of them coupled in "one way". In such cases, a subproblem, B, depends on the results of a subproblem A, without reciprocity. For instance, a bottleneck machine in a job shop is scheduled first and the resulting scheduling influences the optimization of "less-loaded" machines in the job shop. In an RH approach, subproblems correspond to time windows where a subproblem is coupled with the previous one. Thus, given two contiguous subproblems, the first one needs to be optimized first as the resulting schedule has an impact on the second time window.

3. Assignment of operations to time windows

Several heuristics can be studied and developed to assign operations to time windows. Some obvious rules can be derived. For instance, assigning the "x" first operations of each jobs – x being a decision variable, to the first time window. Another approach is to

balance the number of jobs per machine in each time window. Other, more complex rules can be derived and studied.

D. Proposed Approach and Methodology

The focus of this research is to develop scheduling approaches for a large job shop in the event of a system disruption or local disturbances in the production system.

As highlighted in the first section, there are two main phases (Scheduling Phase and Re-Scheduling phase) in this study.

The general approach that will be adopted to solve this problem is as follows:

1. Exploring, adapting, developing and testing scheduling approaches so as to solve a job shop problem for the objective of minimizing the makespan. Two techniques will be explored: the classical SBP and an RH heuristic. After developing theoretical concepts these techniques will be tested on a set of instances.
2. Developing an effective and efficient approach that takes into account disruptions in the production system. To achieve this, the impact and the influence of the three parameters listed in the "Research Hypotheses" (Chapter 1) on the re-scheduling step will be analyzed through several cases and situations. Studying several combinations with various values given to these three parameters may lead to distinguishing some "classes of problems" – each class of problem will motivate a specific scheduling approach. It is likely that these parameters will interact with the decision variables, namely the length of the rescheduling period T1 and the re-scheduling technique used.

4. Results

A. Phase 1: Scheduling with R.H. Heuristics

In this section, two decomposition heuristics are presented and applied to various instances.

1. The General Scheme

The decomposition heuristic (DH) used in this section is straightforward. Its general scheme is shown in figure 7. The idea is to decide on the number of time-windows and to balance the number of operations taken from each job so as to build a time window. If a 10x10 problem (10 machines and 10 jobs) is considered, it can be decided that this instance would be divided into three time-windows: the first time window contains the first three operations of each job, the second time window covers the next three operations of each job, while the third time window includes the four remaining operations of each job.

Once the 10x10 instance is divided into time windows, a Shifting Bottleneck Procedure is applied to the first time window. The completion time of the last operation of job j in the first time window is the release time of the first operation of this job in the second time window.

Once the release dates of the jobs in the second time window are updated according to the optimization of the first time window, the SBP is applied to the second time window. The scheme is repeated iteratively untill the last time window is optimized.

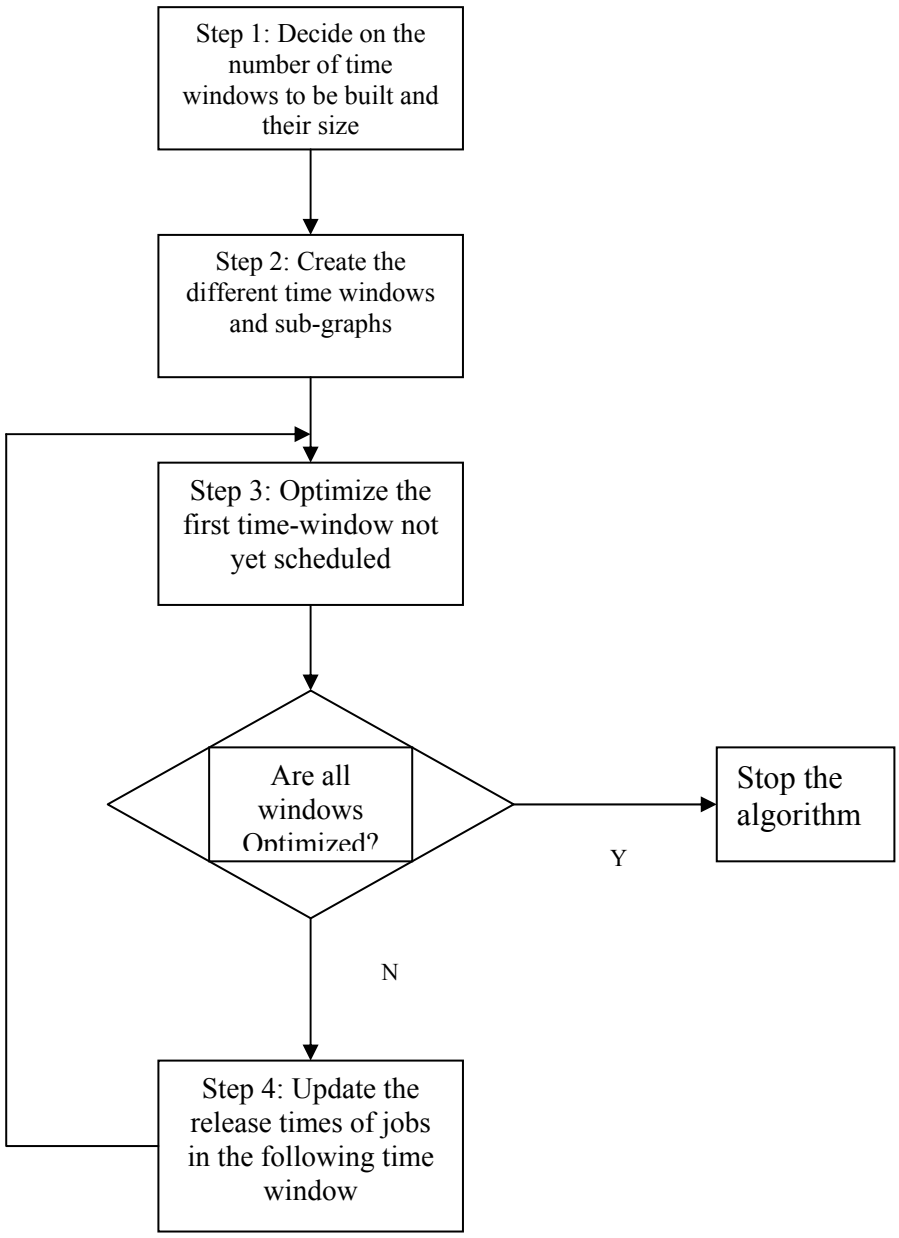

Figure 7 – Decomposition Heuristic general scheme

2. An example

a. Instance

Consider the instance with 4 machines and 4 jobs which routes, i.e., the machine sequence as well as the processing times, are given in table 1:

Jobs	Machine sequence	Processing times
1	1, 2, 3, 4	100, 20, 36, 25
2	2, 1, 3, 4	45, 45, 35, 30
3	3, 2, 1, 4	50, 20, 80, 34
4	4, 2, 1, 3	60, 60, 54, 20

Table 1- A 4 machines, 4 jobs instance

The graph corresponding to this instance (fig. 8) is divided into two time windows.

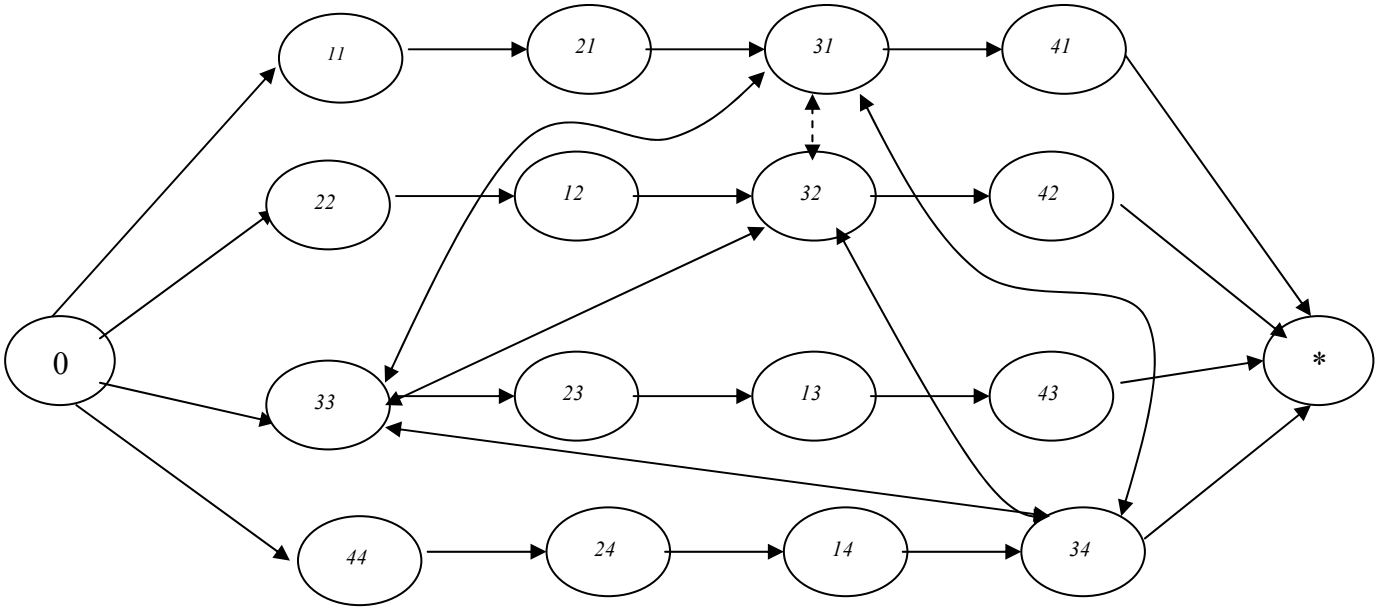

Figure 8 - Overall graph for a 4x4 instance

The SB procedure is applied to the first time window. Figure 9 is a conjunctive graph depicting the time-window after being scheduled.

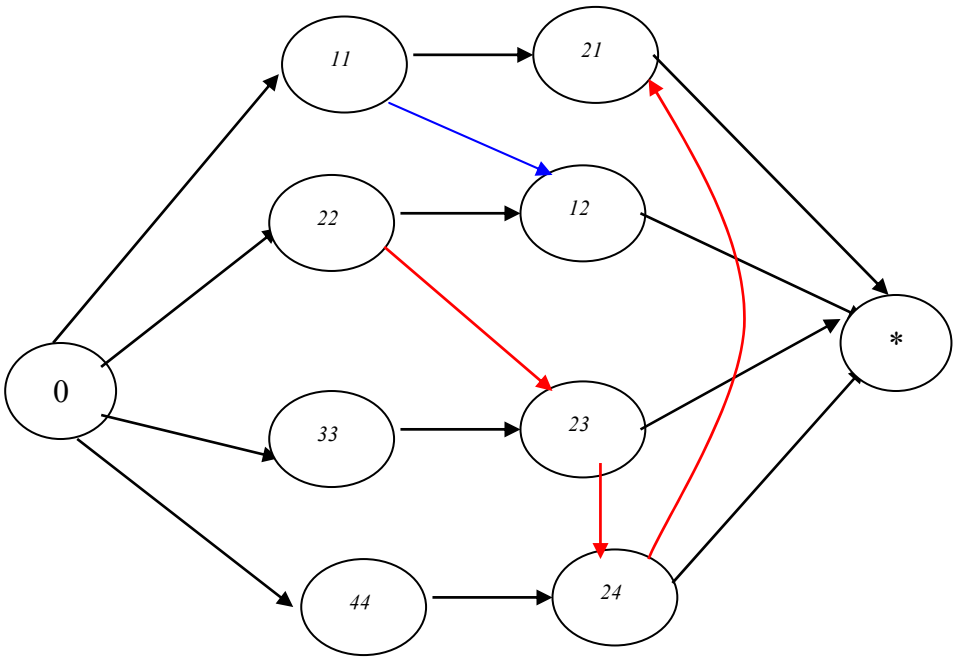

Figure 9 - Graph corresponding to the 1st time-window

Figure 10 is a Gantt Diagram representing the scheduled obtained for the first time window.

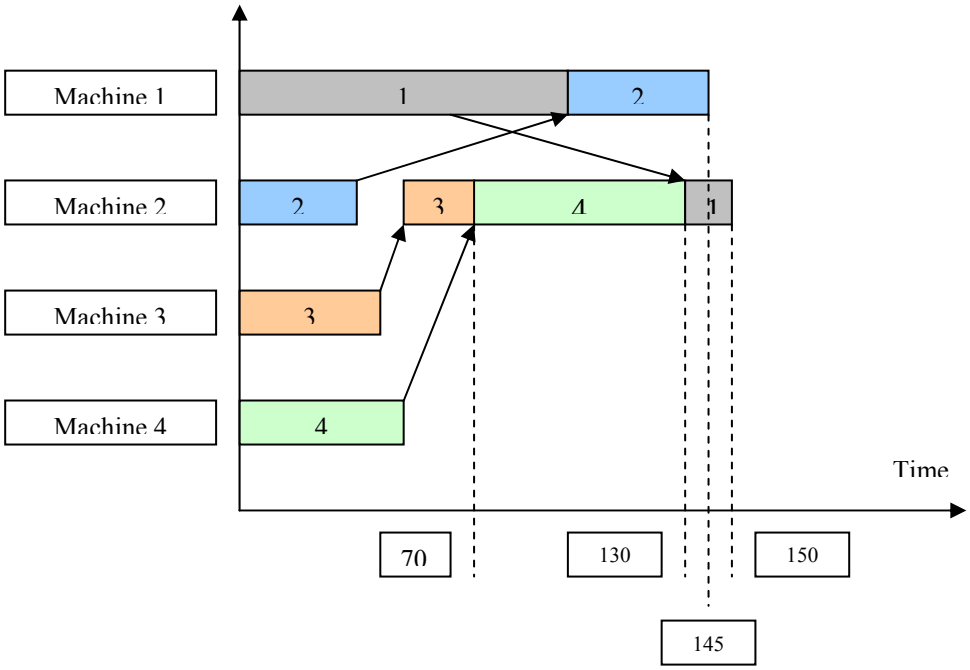

Figure 10 - Gantt Diagram for the first time window of the problem

The transition nodes between the first and the second windows are : (2,1) (1,2) (2,3) (2,4). Their completion time in the first time window are, respectively: 150, 145, 70 and 130. These values, as shown in figure 11, will be used to initialize the graph of the second window. The release time of node (1,4) is the completion time of node (2,4) and so on. The SBP is applied to the second time window.

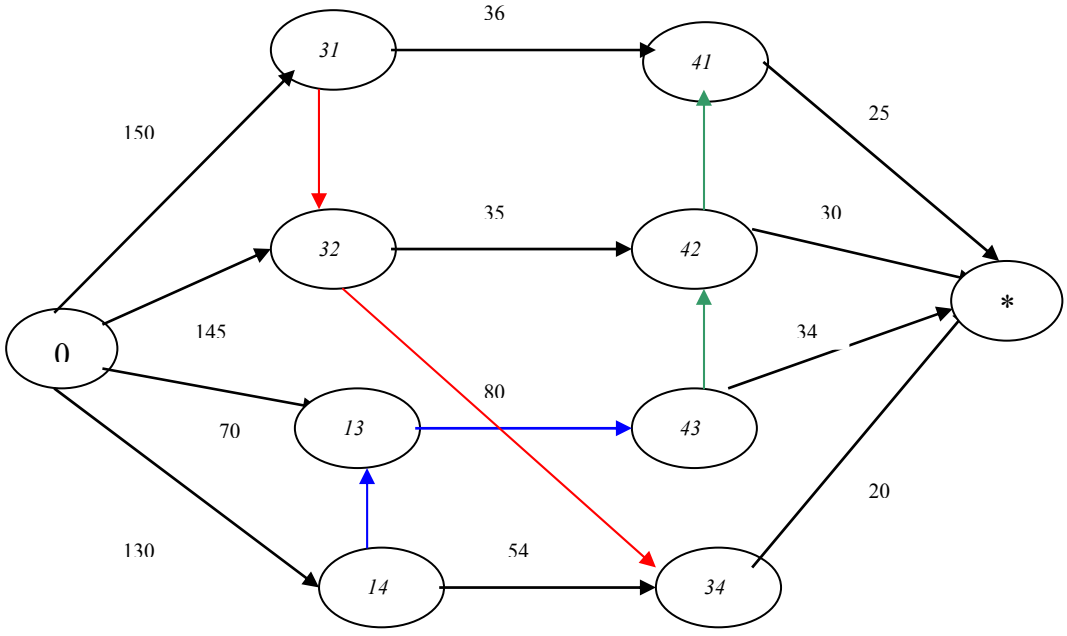

Figure 11 - Graph associated to the second time window

At this stage, all the nodes are scheduled. The solution may be inconsistent (fig. 12), however**,** since, for instance, node (1,4) is supposed to be processed at t = 130 on machine 1, while this machine is not available before t = 145 (i.e, the completion time of node (1,2) in the first window). As a consequence, if the algorithm stops here, the critical path in this graph will be lower than the real critical path of the problem. The value of the critical path at this stage is 353.

b. Updating the Graph

To deal with the overlapping problem depicted in figure 12, the graph should be updated. Three machines are involved in this time-window. By now, all three are scheduled. The first node scheduled on machine 1 is node (1,4). The incoming arc of this node (fig. 11), i.e. the arc between the source and (1,4), has a current value of 130. This value means that job 4 is ready to be scheduled, but it does not guarantee that the machine on which it has to be scheduled – machine 1 is ready.

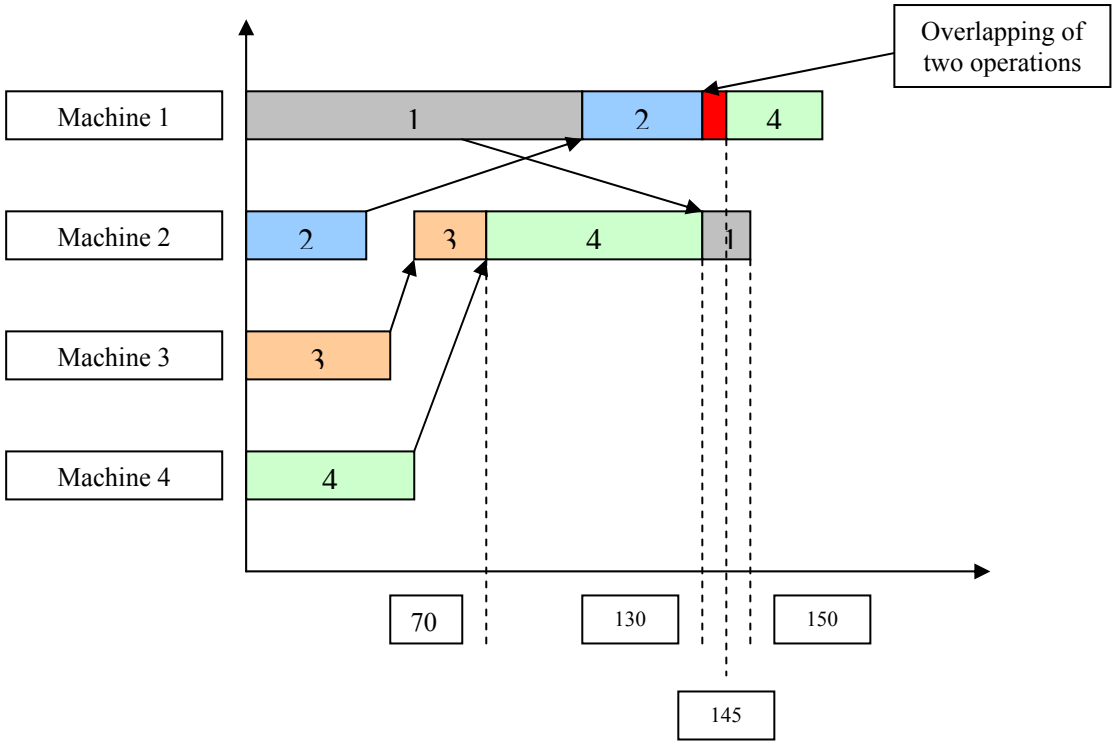

Figure 12 - Gantt diagram showing an overlapping

The weight of the incoming arc of node (1,4) is updated so that the completion time of node (1,2) is considered. The weight, w, of this arc should be modified as follows:

If $C_{m1\text{-}1} > C_{2,4}$, then $w = w + C_{m1\text{-}1} - C_{2,4}$, where : $C_{m1\text{-}1}$ is the completion time of machine 1 in the first time window and $C_{2,4}$ is the completion time of node (2,4), last operation of job 4 done in the first time window. In the instance, $C_{m1\text{-}1} = 145$. Therefore, the new weight of the edge linking the source and node (1,4) becomes 145.

After this update, a realistic solution is obtained as depicted in the Gantt diagram of figure 13. The makespan of the problem is 368.

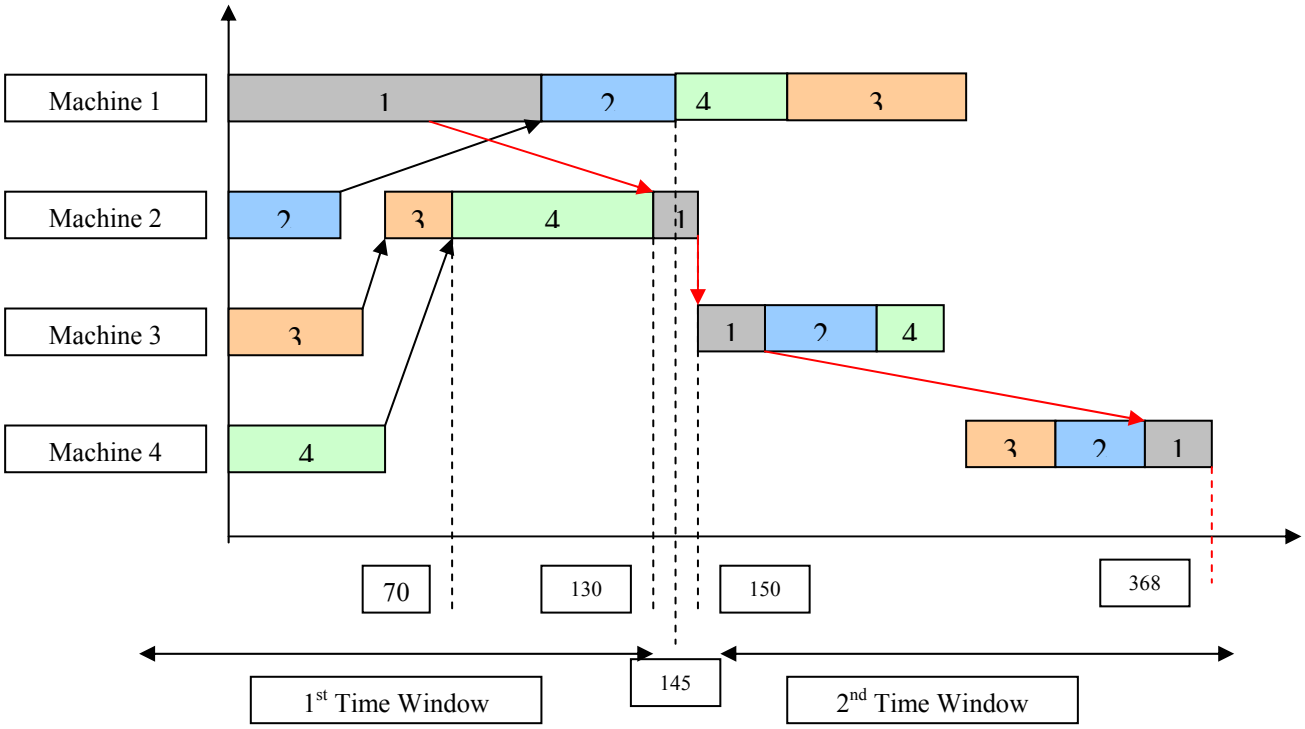

Figure 13 - Gantt Diagram for the Decomposition Heuristic

3. Number of time-windows

The Decomposition Heuristic was applied to several 5x10, 5x15, 5x20 & 10x10 examples. The job shops considered were these used in "Lawrence tests". The impact of the number of subwindows considered in the decomposition on the makespan as well as the time needed to obtain the solution was studied. For the 5x10, 5x15 & 5x20 examples, three cases were considered: SBP applied to whole problem, DH with 2 subwindows and DH with 3 subwindows. For 10x10 cases, 4 tests were done: SB applied to whole problem, DH with 2 subwindows, DH with 3 subwindows and DH with 5 subwindows.

The computational tests are performed on a K6-II 350 MHz, measuring the time in CPU seconds.

- The makespan:

  The first expectation is that the quality of the solution (Cmax) decreases when the number of subwindows involved in the decomposition increases. This was confirmed in most tests. Some examples are given in figure 14.

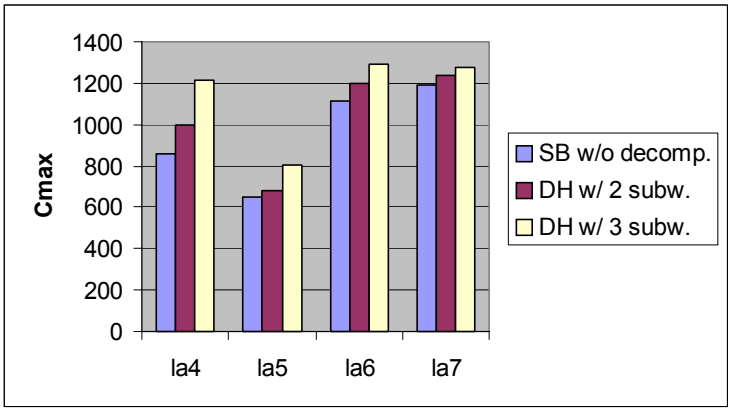

Figure 14 - Impact of the number of subwindows; 5x10 and 5x15 problems

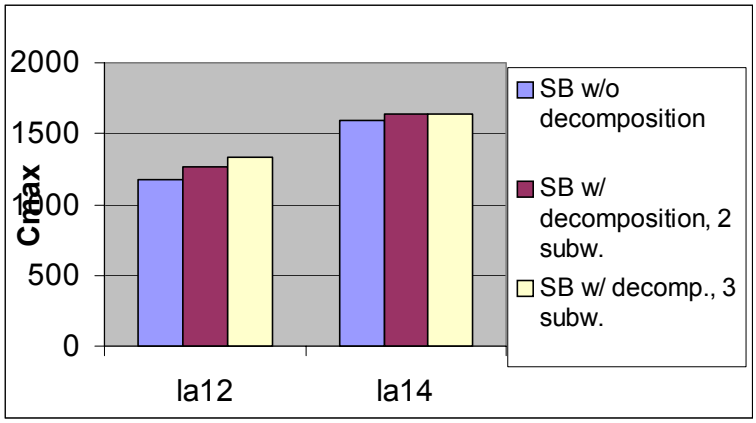

Figure 15 - Impact of the number of subwindows; 5x20 problems

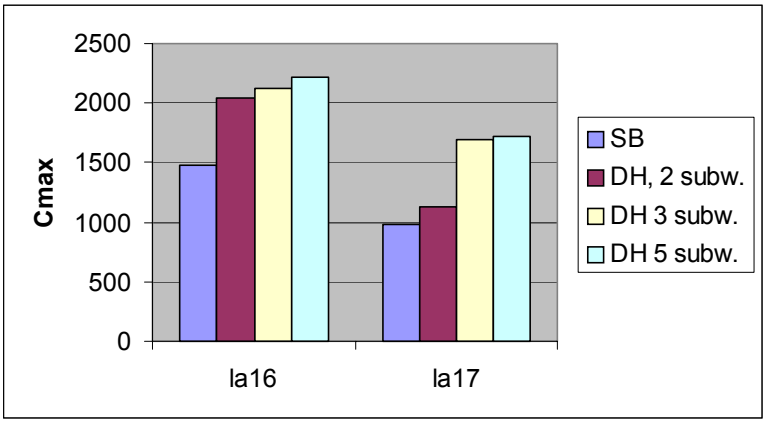

Figure 16 - Impact of the number of subwindows; 10x10 problems

Even if the first expectation is that Cmax increases when the number of subwindows is greater, in some cases, the solution obtained with 3 subwindows is better than the one obtained with 2 subwindows. Figure 17 illustrates this case.

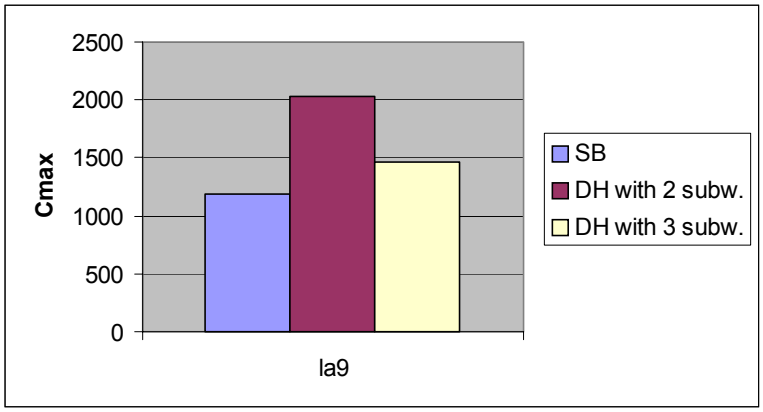

Figure 17 - la-09 instance

Similarly, experiments show that the decomposition approach might retrieve, in few cases, a better solution than the one found by the SB applied without decomposition.

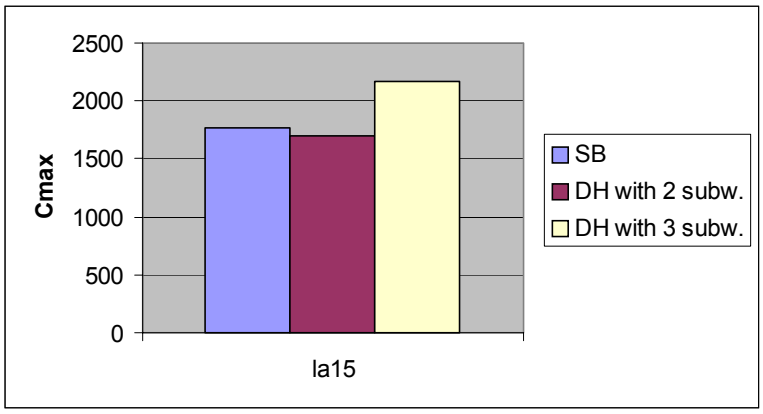

Figure 18 - la-15 instance

These exceptions are due to the fact that Carlier's algorithm is a heuristic for a NP-hard problem; it behaves efficiently in practise, but for some problems, it might give a better solution with a D.H. than with a classical SBP. In addition, in the experiments carried out, the SBP was implemented without the optional re-optimization step. This step is time-consumming, especially if it is applied to semiconductor manufacturing job shops involving more than a hundred machines and a hundred jobs. Thus, to bring useful insights to the seminconductor fabs scheduling, this step was omitted in the experiments carried out on smaller instances. This implies that once a machine is

added to the scheduled set of machines, its schedule is not revised. As a consequence, "deviations" from better solution might be cumulated iteration after another before the algorithm stops.

- Time considerations:

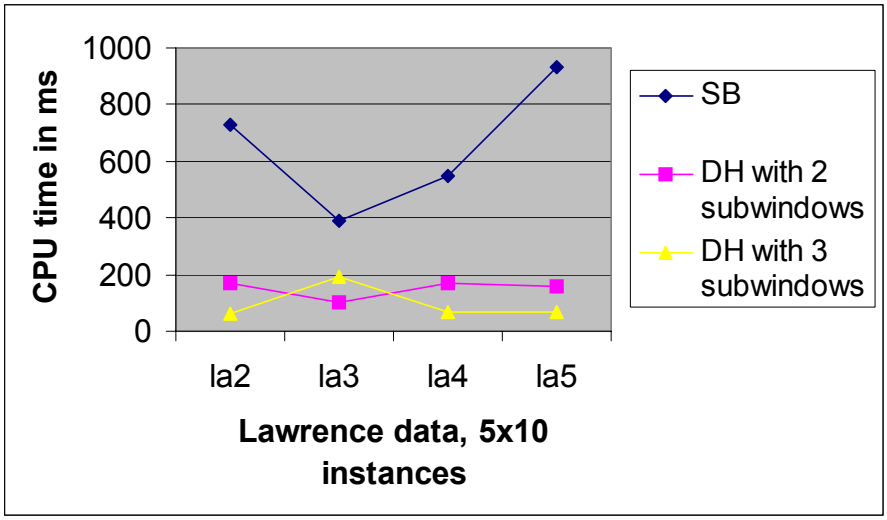

Figure 19 - CPU time for DH for 5x10 instances, according to the number of subwindows

Figure 19 shows the CPU time (in ms) to obtain a solution. It is clear, and expectable, that the CPU time decreases when the number of subwindows increases. This is also confirmed for 5x15, 5x20 & 10x10 instances, as depicted in figures 20 & 21, & table 2.

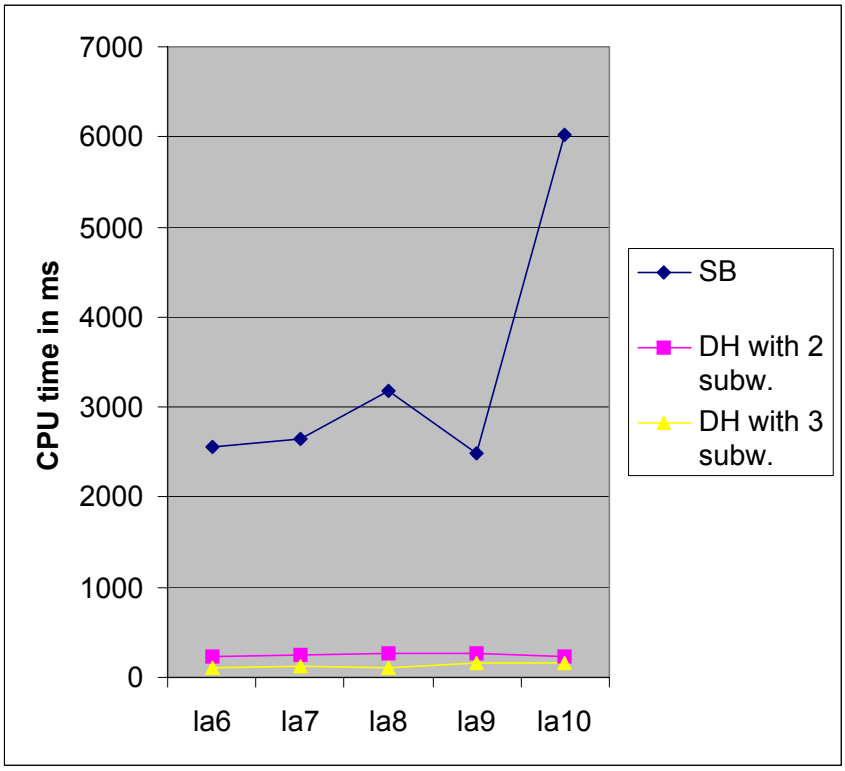

Figure 20 - CPU time for DH for 5x15 instances, according to the number of subwindows

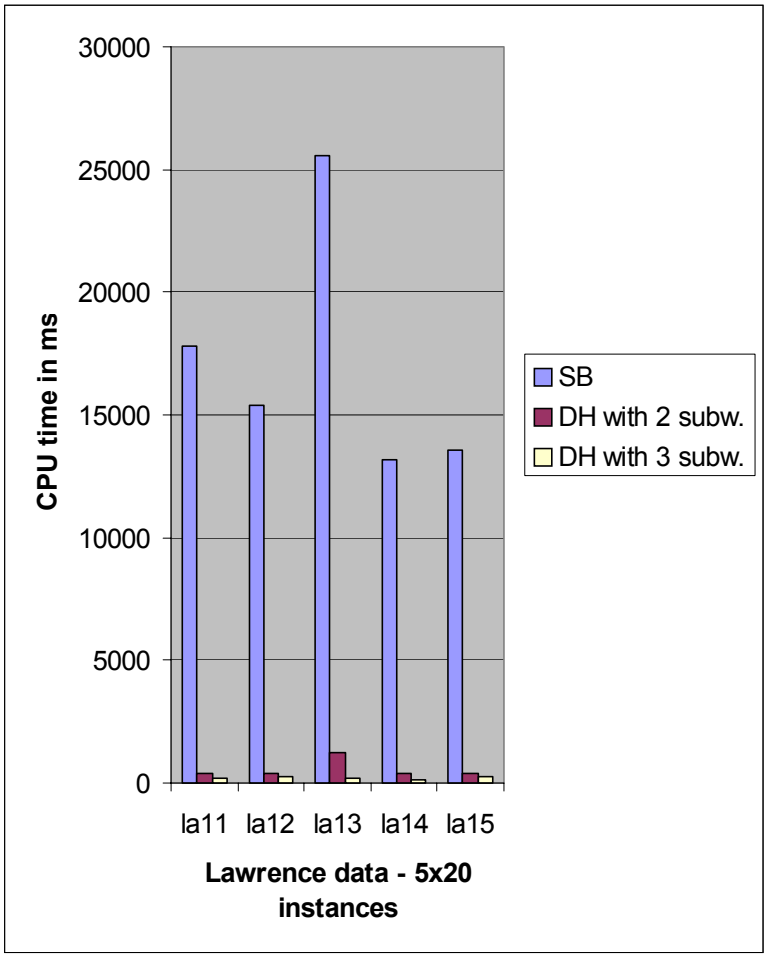

Figure 21 - CPU time for DH for 5x20 instances, according to the number of subwindows

	SB, t in ms	DH, 2 subw., t in ms	DH, 3 subw., t in ms	DH, 5 subw., t in ms
La16	19438	631	380	180
La17	16133	601	381	180
La18	30023	681	351	100
La19	11427	690	515	231
La20	25505	671	351	260

Table 2 - CPU time for DH for 10x10 instances, according to the number of subwindows

4. Improved Decomposition Heuristic (IDH)

Instead of considering different sub-graphs - each representing a sub-window of the problem - only one general graph is considered. The optimization of each sub-window is done on the general graph.

- Schedule the first window:

Nodes corresponding to the first time-window will be scheduled using a SBP. The corresponding edges will be included in graph. When a machine is added, the graph is updated.

- Schedule the second window:

While some machines are not scheduled yet:

→ Using Carlier's algorithm, select the machine causing the severest disruption in the current window.

→ Add the corresponding edges to the graph.

→ Check if this machine is present in the first time window. If this is the case, link the first node to be scheduled on this machine in the second window to the last node scheduled on this machine in the previous window. This approach guarantees the absence of overlapping.

→ Update the whole graph.

This method is more precise than the previous one. In fact, an update is done after adding any machine, and this update considers the whole graph, not just the nodes to be scheduled in the current time window.

In the example of figure 22, nodes in the first time window were scheduled on the appropriate machines.

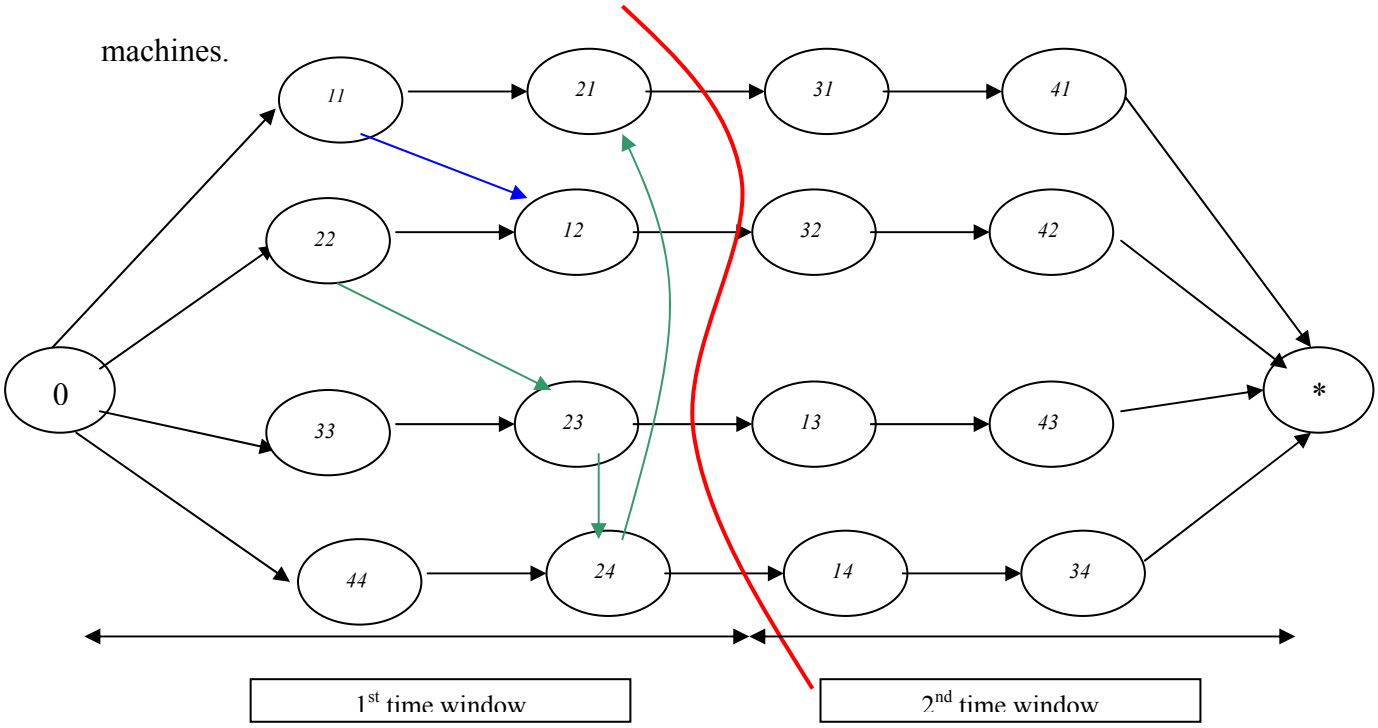

Figure 22 - Construction of the solution with the IDH

The scheduling of the second time window is now considered. Assume the machine 1 is scheduled in the second time window, so that operation (1,3) is done before operation (1,4). Two arcs will be added: one, linking (1,3)→ (1,4) and the other linking (1,3) to the last operation done on machine 1 in the previous time window: (1,2) → (1,3).

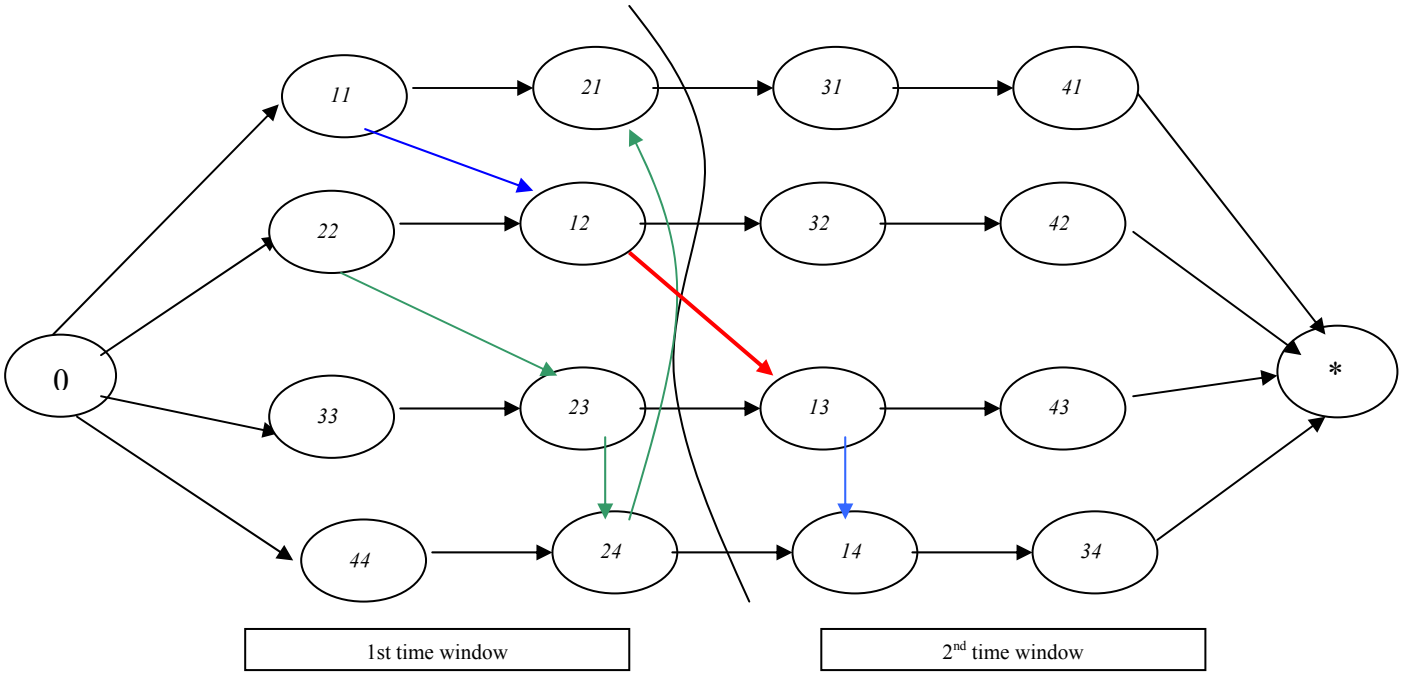

Figure 23 - Updating the nodes in the second time window

5. Some Results

The following approaches were tested on various instances:

- A Shifting Bottleneck Procedure applied to the whole graph.
- The decomposition heuristic using a SB procedure (as described previously).
- The Improved Decomposition Heuristic (IDH)

The computational tests are performed on a K6-II 350 MHz, measuring the time in CPU seconds. Two time-windows were considered in decomposition methods for 5x10, 5x15 and 5x20 instances, whereas the 10x10 problems were divided into three time windows when decomposition methods were applied.

a. 5x10 problems

This graph shows that the SBP outperforms the two decomposition methods. It is interesting to note that IDH reduces the gap between the SB w/o Decomposition and the first Decomposition Heuristic.

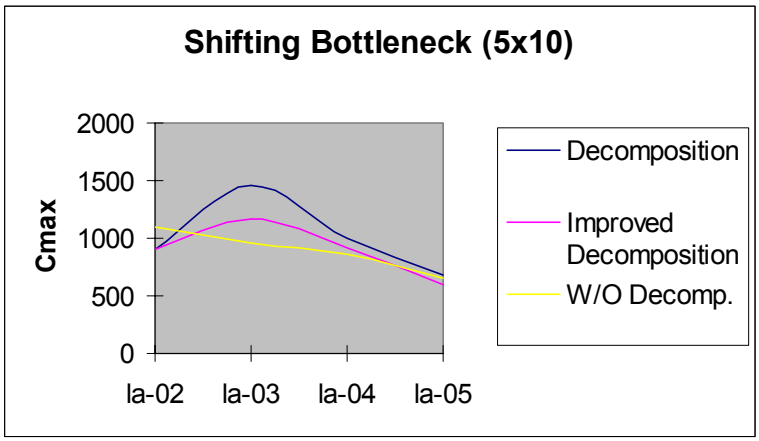

Figure 24 - Comparison between three heuristics for 5x10 problems

b. 5x15 problems

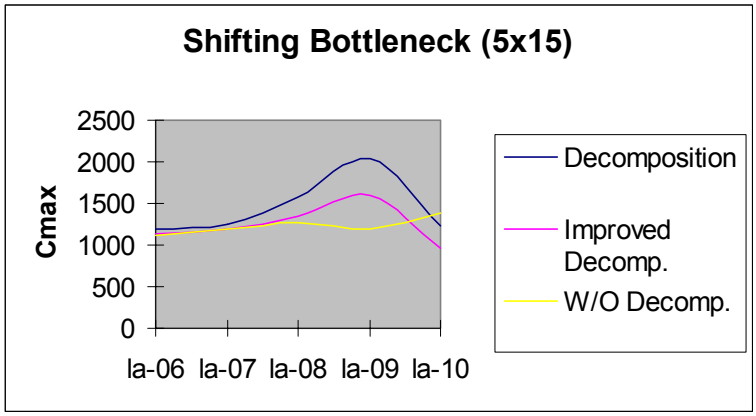

Figure 25 - Comparison between three heuristics for 5x15 problems

The SBP finds a better solution overall. For the first two instances (la-06 and la-07), the three approaches retrieve similar solutions. The SBP, however**,** outperforms the decomposition methods for la-08 and la-09. The IDH gives intermediary results and reduces the gap between the SB without decomposition and the first decomposition heuristic.

c. 5x20 problems

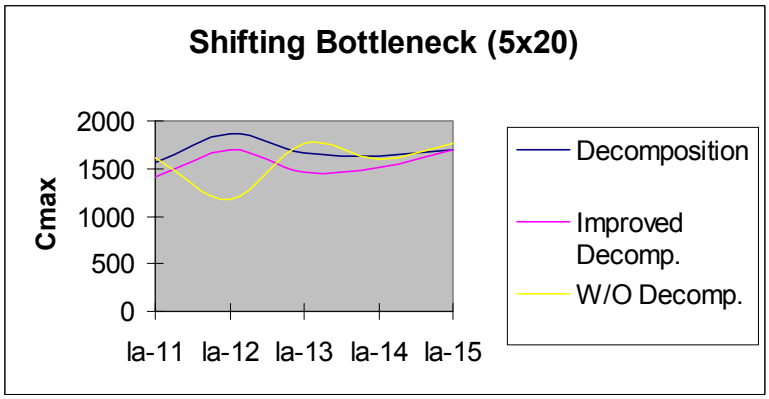

Figure 26 - Comparison between three heuristics for 5x20 problems

The IDH outperforms the first decomposition heuristic. It even outperforms the SB procedure applied without decomposition. It is interesting to note that the SB procedure applied without decomposition clearly outperforms the decomposition methods for the la-12 instance.

d. 10x10 problems

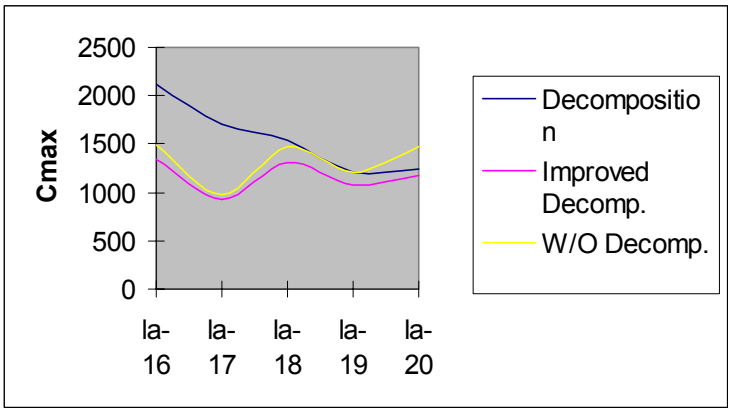

Figure 27 - Comparison between three heuristics for 10x10 problems

The IDH generally outperforms the other approaches. The difference between the IDH and the SB is quite negligible for the first 2 instances, whereas the DH is clearly outperformed for these instances. However, the gap is rather reduced between the 3 approaches for la-18, la-19 and la-20. The SB is outperformed by the decomposition heuristics for the last three instances.

e. Time Considerations

It is clear that the DH is much faster than the other approaches. It is also quite expected to see that the SB applied without decomposition takes more time than the Improved Decomposition Heuristic.

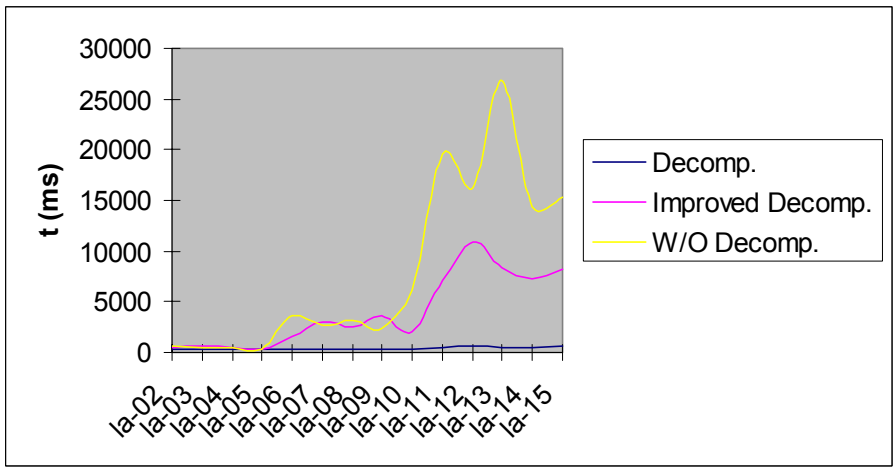

Figure 28 - CPU time for 5x10, 5x15 and 5x20 problems

B. Phase 2: Re-Scheduling Phase

1. Introduction

In this section, re-scheduling approaches in the event of a breakdown are studied. It is assumed that a schedule was obtained in Phase 1. This schedule is applied in the fab. At time t, an unexpected breakdown or production disturbance occurs on a given machine. A set of operations has been scheduled on the different machines of the job shop between time 0 and t. For the operations that have not been processed yet, the scheduling of Phase 1 needs to be re-considered.

In this dynamic re-scheduling study, three parameters are of interest:

- The time t at which the breakdown occurs. This parameter has a direct consequence on the number of operations that have been processed before the breakdown occurs (see the Gantt Diagram below).
- The length of the breakdown, L.
- The machine on which the breakdown or the production disturbance has occurred.

The re-scheduling can be done on the whole set of remaining nodes. It might be interesting however, to do the re-scheduling on a limited horizon. As a consequence, the set of nodes that have not been processed yet will be divided into two sub-sets. The first sub-set will be re-scheduled, while the second one will be scheduled according to the schedule generated in phase 1.

In Phase 1, a Modified Shifting Bottleneck is applied to the problem. A schedule is generated.

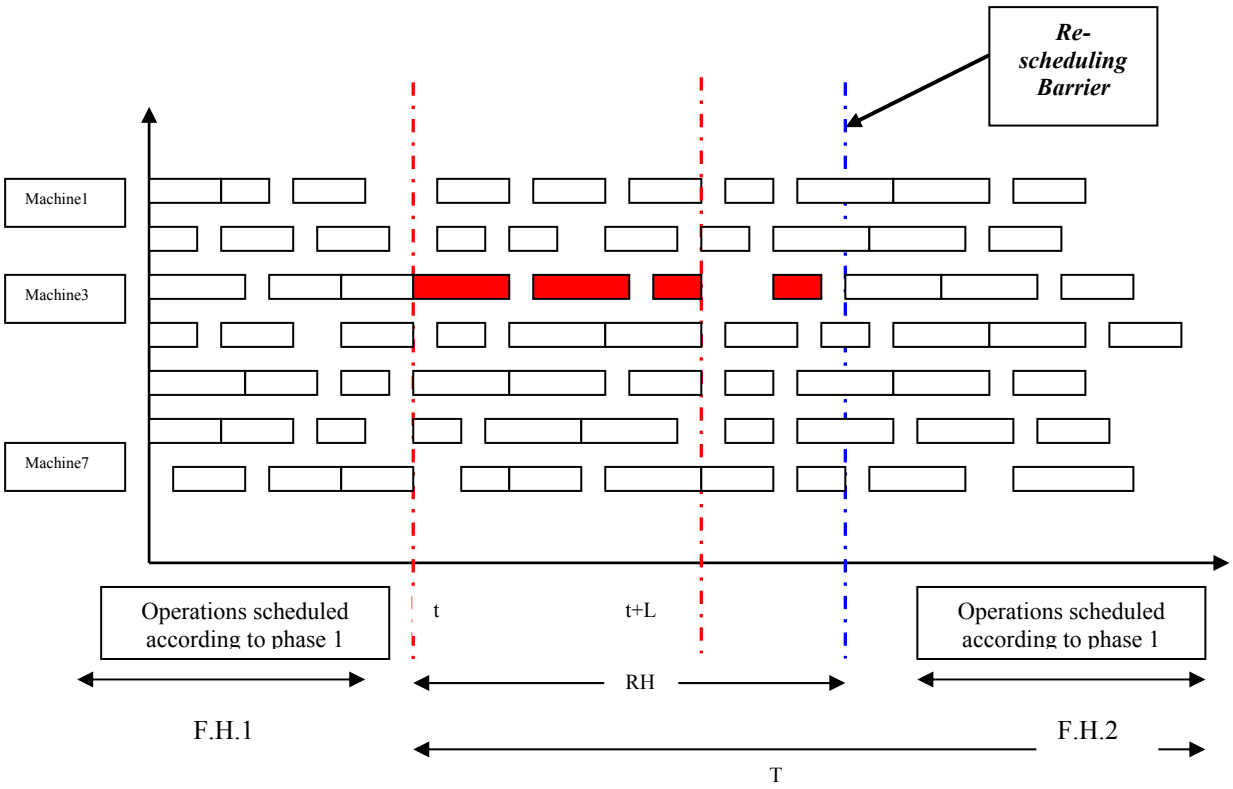

Figure 29 - Gantt Diagram before re-scheduling

2. Notations and hypothesis

- t: time at which the breakdown occurs on a machine.
- L: the length of the breakdown, known as soon as the breakdown occurs.
- R.H.: rescheduling horizon.
- F.H.: horizon fixed according to the schedule obtained in Phase 1.
- T: Total potential rescheduling horizon.
- The length of R.H. varies between L and T – t –L.

The machine affected by the breakdown or the local disturbance, machine 3, is shown in red in the graph as well as the Gantt diagram.

The operation which is processed when the breakdown occurs, is assumed to be completed and therefore is not included in the set of operations to be re-scheduled.

3. General Scheme

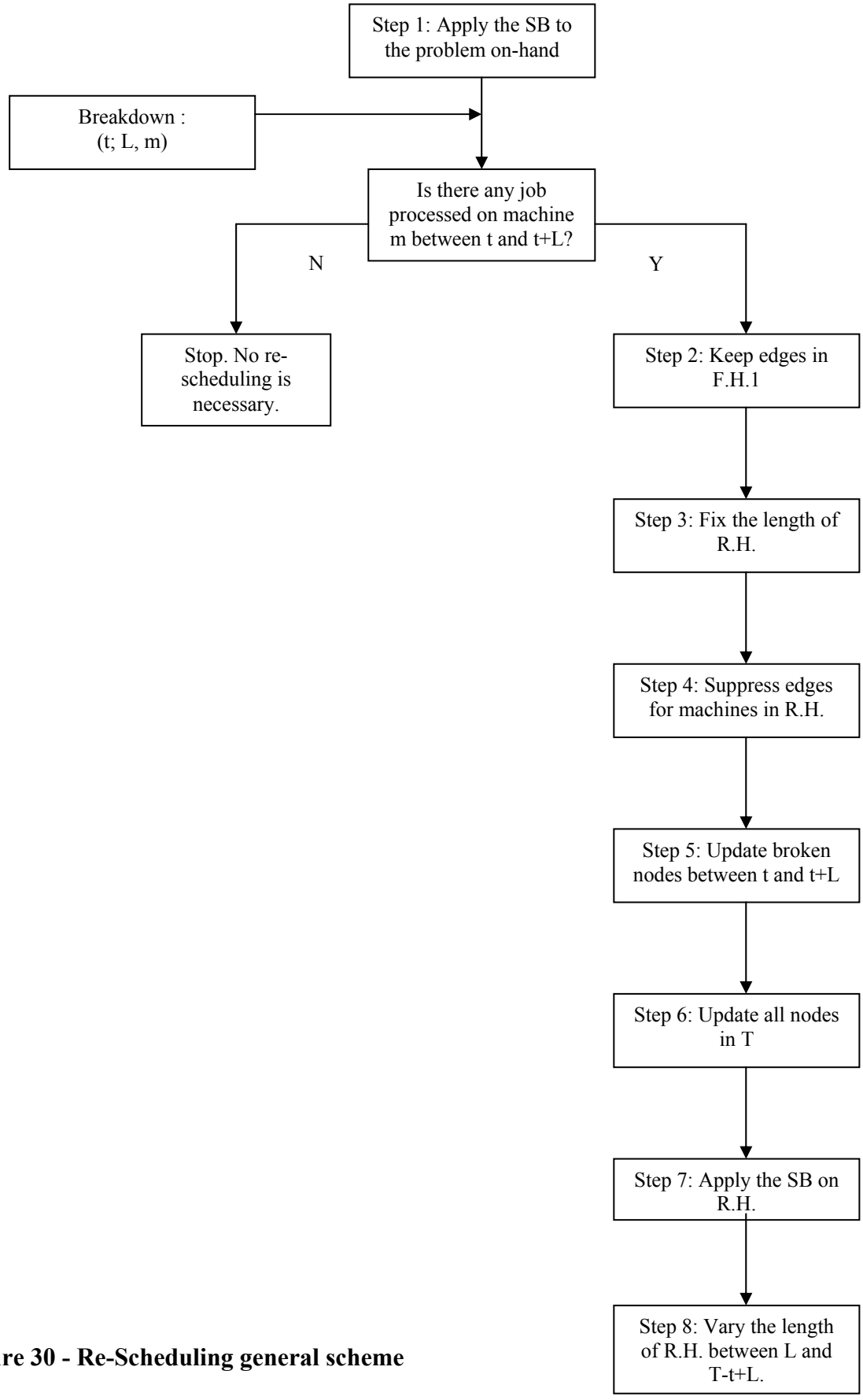

Figure 30 - Re-Scheduling general scheme

In step 4, the SB is going to be applied to the subgraph representing the R.H. portion. Therefore, in this subgraph, only conjunctive arcs linking the successive operations of a job are kept. Conjunctive arcs resulting from the schedule done in Phase 1 (before the breakdown) are deleted in R.H. whereas they are kept in F.H. 1 since this portion of the graph is not re-scheduled.

In step 5, the unavailability of the machine on which the breakdown occurred between t and t+L has a consequence on the start time of the operations that initially (in phase 1) could have been scheduled between t and t+L. These operations can no more be scheduled during this period and this constraint should be integrated in their release time. This is done in step 5. The update is then propagated in the rest of the graph in step 6.

At this stage, the graph is updated, the R.H. is well defined, and the SB is applied to re-schedule R.H. in step 7.

To appreciate the impact of the impact of the length of R.H. on the new schedule, this length shrinks progressively in step 8.

4. An example

Consider a 4x4 job shop problem. The routes, that is, the machine sequencesand the processing time, are given in the following table:

Jobs	Machine sequence	Processing times
1	1, 2, 3, 4	100, 20, 36, 25
2	2, 1, 3, 4	45, 45, 35, 30
3	3, 2, 1, 4	50, 20, 80, 34
4	4, 2, 1, 3	60, 60, 54, 20

Table 3 - A 4x4 instance

Step 1: Applying the SB to this job shop. The solution built is represented by the following graph.

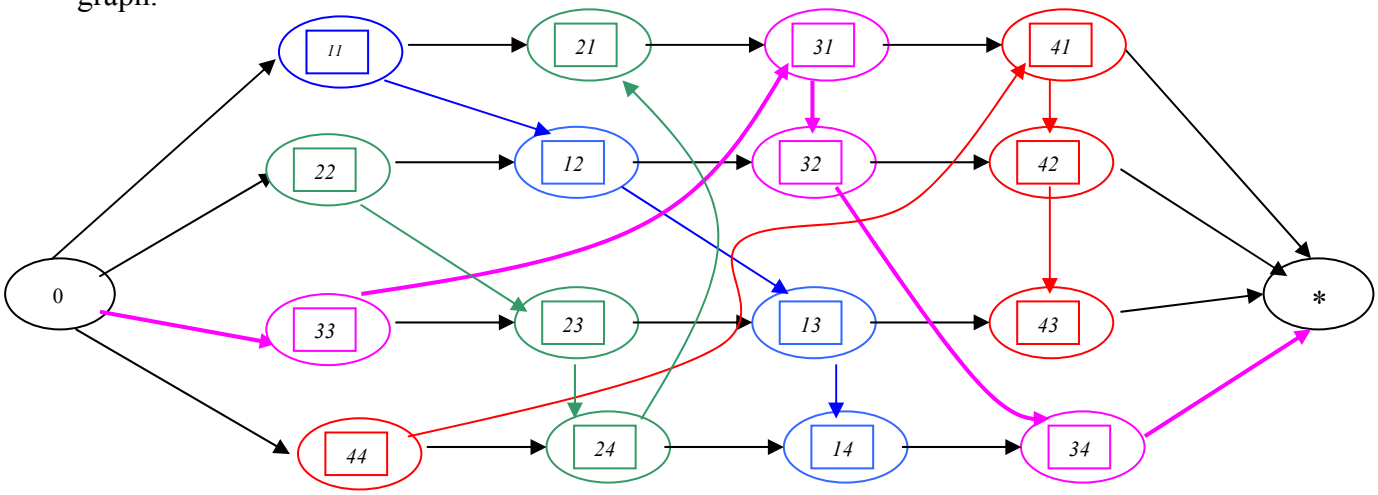

Figure 31 - Scheduled Graph with a SB procedure

The makespan obtained for this schedule is 299. The Gantt siagram for the problem is:

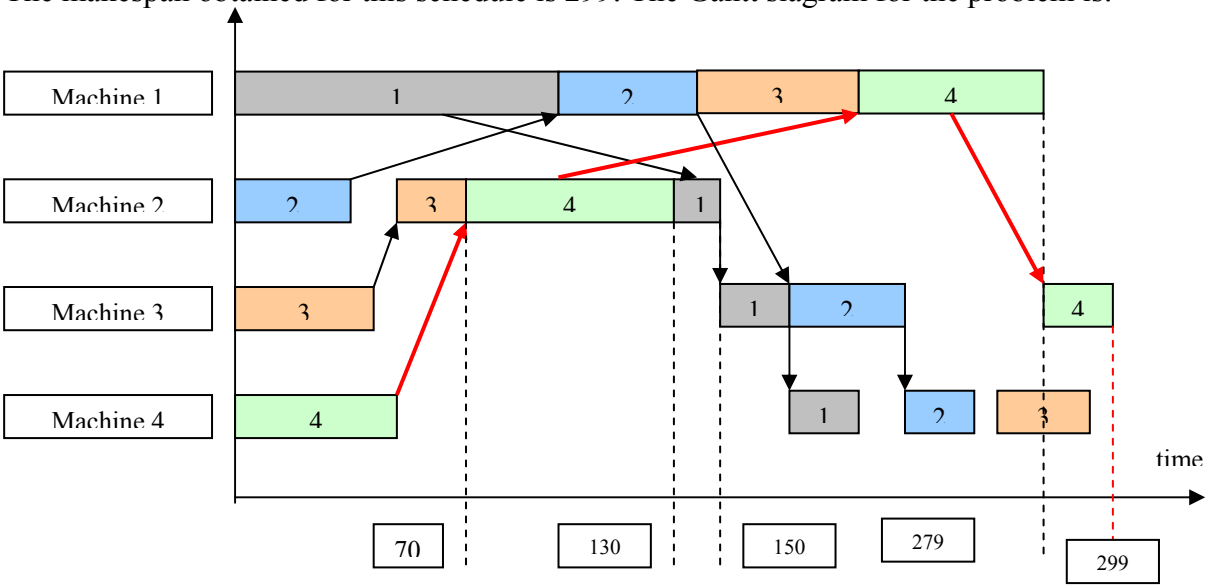

Figure 32 - Gantt Diagram for a 4x4 instance

Once this schedule is obtained, it is applied in the shop. However at time t = 120, an unexpected breakdown occurred on machine 2. We assume that as soon as this breakdown occurred, its estimated length (L=50) can be determined by the maintenance personnel for instance. According to the schedule in phase 1, some jobs were supposed to be scheduled on machine 2 between t =120 and t = 170. Therefore, a re-scheduling is necessary.

Steps 2, 3 and 4: Since the operations in this time-window have been processed, arcs in F.H.1 are kept. The re-scheduling will be done for all of the remaining nodes.

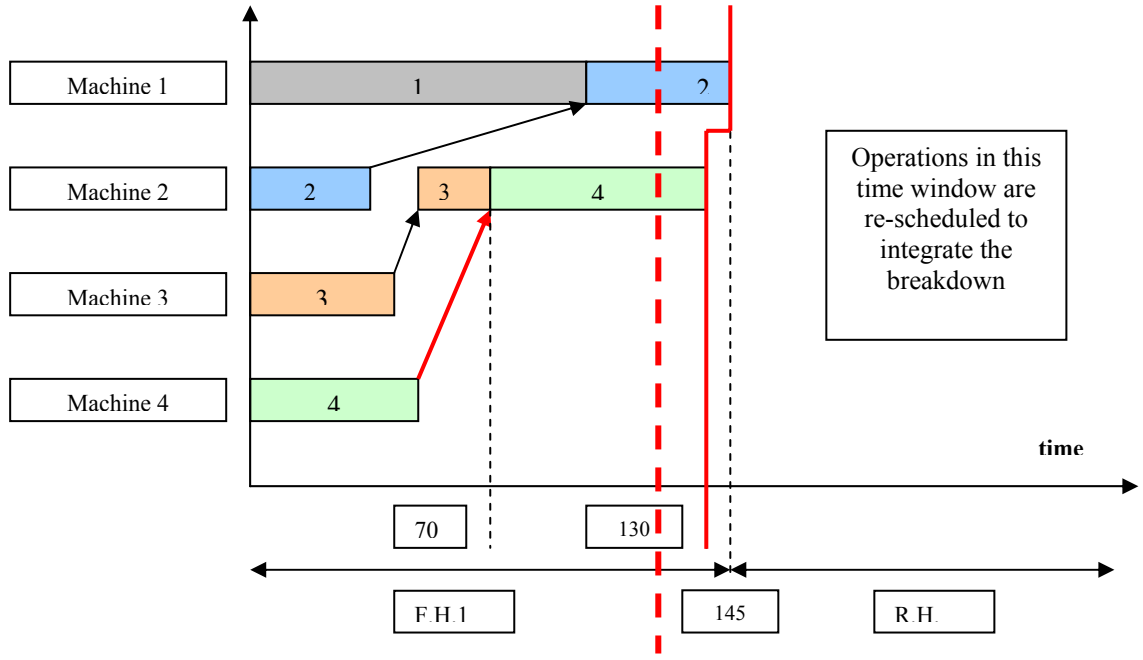

Figure 33 - Reconstruction of the schedule to deal with a breakdown

Step 5: Figure 34 illustrates the state of the graph at this stage.

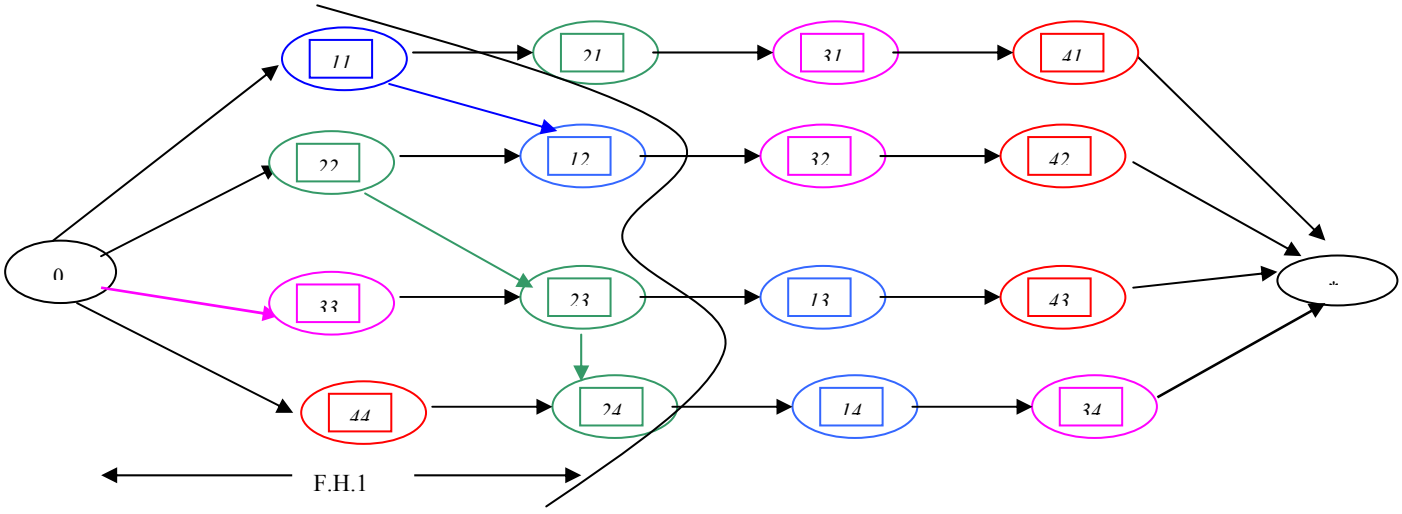

Figure 34 - Reconstruction of the graph to deal with a breakdown

The SBP is applied to the R.H. and the makespan of the problem is 319, while the makespan would be 299 if no breakdown occurred.

If, however, this breakdown (t= 120 and L =50) had occurred on machine 4, the makespan of the problem would still have been 299. In fact, the re-scheduling procedure tests whether any job is scheduled between t = 120 and t = 150 on machine 4. Since the answer is no, no re-scheduling is needed and the heuristic stops and the schedule obtained in phase 1 is kept.

According to the schedule obtained in phase 1 (see figure 16), the following general comments can be derived:

- Machine 4: for any breakdown (t,L) on machine 4, such that t > 60 and t+L < 170, no rescheduling is needed.
- Machine 3: for any breakdown (t,L) on machine 3, such that t > 50 and t+L < 150, no rescheduling is needed.
- Machine 2: for any breakdown (t,L) on machine 2, such that t > 150 no rescheduling is needed.

Variations of t:

Some breakdowns were simulated with (L,m) fixed and t varying. Put in other words, the time where the breakdown occurs on machine m varies, while its length does not.

- (L,m) = (50, 4).

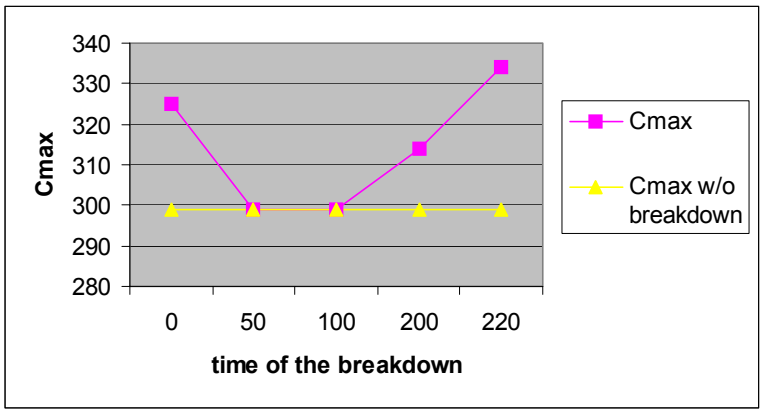

Figure 35 - Cmax for (L,m) = (50,4)

For t = 50 and t =100, no re-scheduling is needed since the breakdown occurs in a time window where the machine is supposed to be idle (see Gant diagram in figure 32).

- (L,m) = (50, 3).

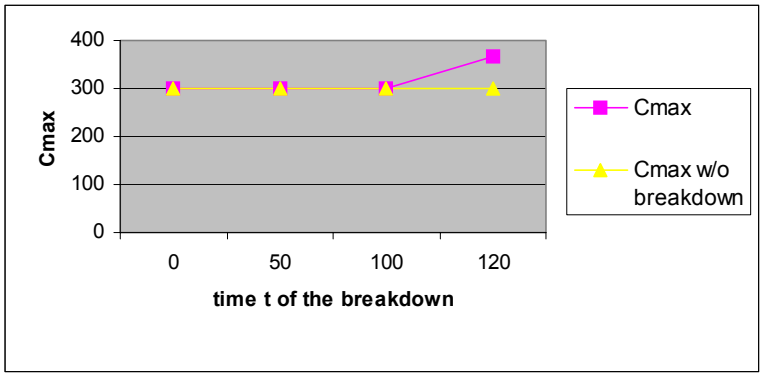

Figure 36 - Cmax for (L,m) = (50,3)

5. Length of R.H.

The re-scheduling procedure, described in figure 30, was applied on different examples. Step 8 of this procedure consists of varying the length of the rescheduling horizon, as explained in figure 29. It is expectable to have a deterioration of the quality of the solution when R.H. shrinks, since reducing the R.H. implies reducing the number of re-scheduled operations and increasing the number of operations fixed according to the schedule obtained in phase 1. It is also quite expectable to have shorter CPU times when R.H. shrinks, since the sub-problem on which the SB is applied in step 7 (figure 30) is smaller. These expectations were confirmed by the various tests that have been carried out. By applying the procedure in figure 30, the user can vary the parameters (t, m, L) as well the length of R.H., depict the makespan variations and the CPU time associated to each configuration, to eventually find trade-offs that meet his needs.

Some examples:

- la02; (t, m L) = (60,2,70) : breakdown at t = 60 on machine 2. the breakdown took 70 time units.

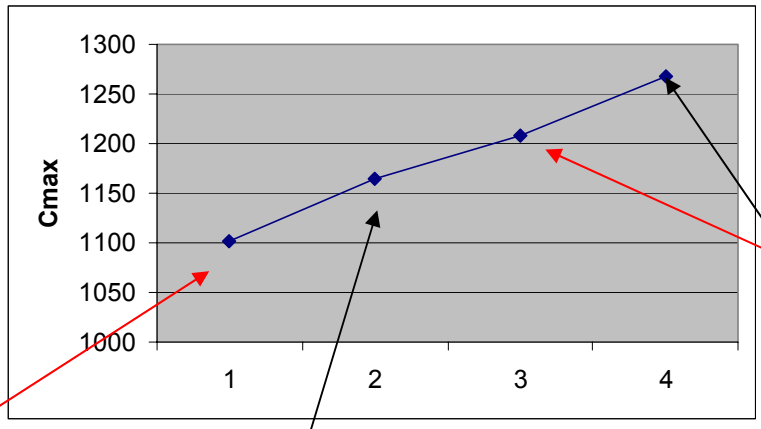

Figure 37 - Cmax for R.H. shrinking, la-02 instance

R.H. shrinking

R.H. = the total potential re-scheduling horizon

The example of figure 35 shows that reducing the rescheduling horizon deteriorates the solution. It also shows that, despite the fact that the breakdown occurred at the beginning of the schedule – t = 60 while the initial makespan is around 1100 -, the re-scheduling phase did not cover up the effects of the breakdown. In fact, if a breakdown occurs at the beginning of the schedule on a machine that is not critical, re-scheduling the shop can absorb the breakdown effects. However, if the breakdown affects a critical machine, it is more likely that the re-scheduling phase will not cover up the lateness generated by the breakdown.

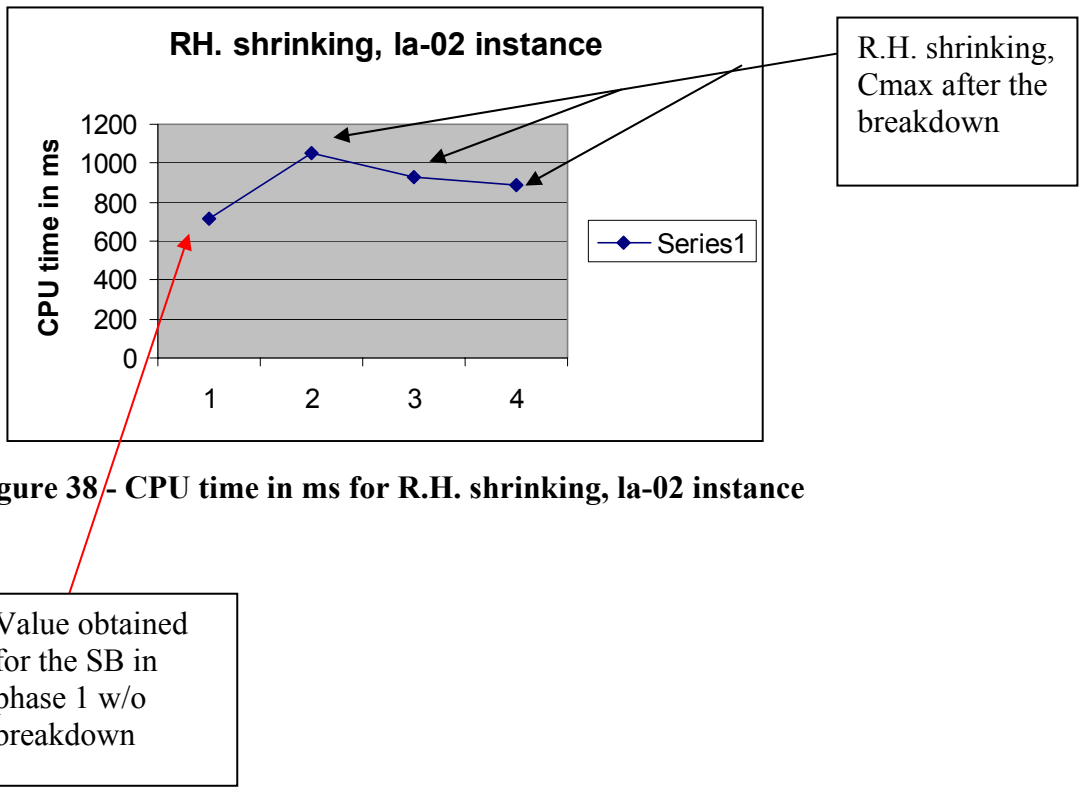

Figure 38 - CPU time in ms for R.H. shrinking, la-02 instance

Figure 36 and figure 37 reflect the decrease of the CPU time when R.H. shrinks and the increase of the Cmax.

- la06 ; (t,m,L) = (60,3,300). breakdown at t = 60 on machine 3. the breakdown took 300 time units.

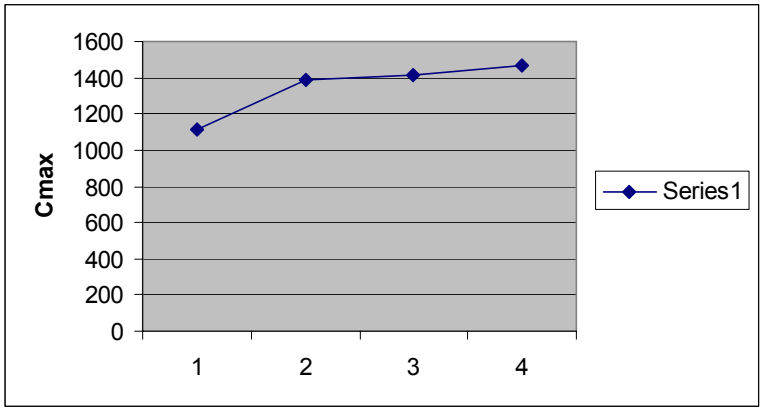

Figure 39 – Cmax for R.H. shrinking; la06 instance, (t,m, L) = (60, 3, 300).

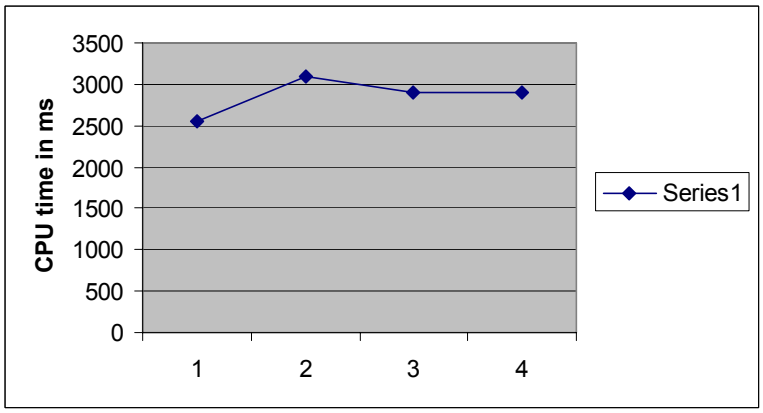

Figure 40 – CPU time for R.H. shrinking; la06 instance, (t,m,L) = (60,3,300)

Reducing the length of the re-scheduling horizon deteriorates the quality of the solution but gives better CPU time. The user should find a trade-off that suits his needs. In addition, if the breakdown occurs early in the schedule and does not affect critical machines, it is possible to cover up its effects in the re-scheduling phase by considering a restricted horizon. However, if critical machines are impacted, reducing the re-scheduling horizon deteriorates the solution

and considering the whole potential re-scheduling horizon (i.e. every unprocessed operation) should be considered.

Conclusion

Scheduling problems that arise in the Semiconductor Industry are very complex. The complexity is due to the numerous constraints inherent to this industry such as the presence of re-entrant flow, parallel and batching machines, … The problem becomes even more complex because production disturbances due to breakdowns.

In the first part of this research, the static job shop problem was considered. Several approaches were implemented. The SB procedure was compared to a straightforward decomposition heuristic (DH). The impact of the number of subwindows built in the DH was analyzed. In addition, an IDH (improved decomposition heuristic) was developed and implemented. It gave interesting results in terms of makespan value and CPU time.

In the second part, reactive scheduling after an unexpected breakdown was studied. A general re-scheduling scheme was presented to deal with these production disturbances. This procedure allows the user to estimate the impact of the length of the re-scheduling horizon on the Cmax and the CPU time to build the solution. The user can then choose a trade-off that meets his needs.

So far, the results and approaches discussed in this research are not restricted to the Semiconductor Industry. They can be explored for productions modeled as job shops. Future extensions can be made to include constraints that arise in the semiconductor industry.

APPENDIX: Elements of Code Implemented in JAVA language

```
//package stagefi4;

import java.util.*;
import java.io.*;

import salvo.jesus.graph.*;

/**
 * <p>Title: Stage de fin d'études</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Ecole des Mines de Nantes</p>
 * @author Ahmad Ghoniem
 * @version 1.0
 */

public class Scheduler {

 public Scheduler() {
 }

  public int minr (Vector v) {

   int minr = Integer.MAX_VALUE;
   for (int i = 0 ; i < v.size(); i++){
     if ( ((Node)v.elementAt(i)).getRel_time() < minr ) {
       minr = ((Node)v.elementAt(i)).getRel_time();
     }
   }

   return minr;
 }

 public Vector supminr (Vector v, int minr) {
   Vector v0 = new Vector ();
   for (int i = 0; i < v.size(); i++){
     Node n = ((Node)v.elementAt(i));
     if ( n.getRel_time() <= minr) {
       v0.addElement(n);
     }
   }
 return v0;
 }

 public Node node_qmax (Vector v, int minr) {
   Vector v0 = supminr(v,minr);
   Node n = (Node)v0.firstElement();
   for (int i = 0 ; i < v0.size(); i++){
     if ( ((Node)v0.elementAt(i)).getQ() > n.getQ()) {
       n = (Node)v0.elementAt(i);
     }
   }
 return n;
 }

 public Vector schrage (Vector nodes) {
   Vector temp = (Vector)nodes.clone();
   Vector scheduled = new Vector();
   int t = minr(nodes);
   while ( temp.size() > 0){
     Node temp_node = node_qmax(temp, t);
     scheduled.addElement(temp_node);
     temp.removeElement(temp_node);
     t = Math.max (t + temp_node.getProc_time(), minr(temp));
   }
   return scheduled;
 }
```

```
public void affiche (Vector scheduled) {

  for (int i=0; i < scheduled.size(); i++) {
    ((Node)scheduled.elementAt(i)).print();
  }
}

public Vector crit_path (Vector scheduled) {


  Vector crit = new Vector();
  Node start = (Node)scheduled.firstElement();
  Node end;
  int slength = start.getRel_time()+ start.getProc_time();
  int index1 = 0;

  for (int i = 1; i < scheduled.size(); i++) {
    if ( slength < ((Node)scheduled.elementAt(i)).getRel_time()) {
      start = (Node)scheduled.elementAt(i);
      index1 = i;
      slength = start.getRel_time()+ start.getProc_time();
    }
    else {
      slength = slength + ((Node)scheduled.elementAt(i)).getProc_time();
    }
  }

  end = start;
  int index2 = 0;
  int elength = 0;
  for (int i = index1 +1 ; i < scheduled.size(); i++) {
    if ( end.getQ() <= elength + ((Node)scheduled.elementAt(i)).getProc_time()
                  + ((Node)scheduled.elementAt(i)).getQ() ) {
      elength = 0;
      end = (Node)scheduled.elementAt(i);
      index2 = i;
     }
    else {
      elength = elength + ((Node)scheduled.elementAt(i)).getProc_time();
    }
  }
  crit.addElement(new Integer (index1));
  crit.addElement(new Integer(index2));
  return crit;

}

 public Node findc (Vector crit, Vector scheduled) {
   int i1 = ((Integer)crit.firstElement()).intValue();
   int i2 = ((Integer)crit.lastElement()).intValue();
   Node c = null;
   for (int i = i1; i < i2; i++) {
      if ( ((Node)scheduled.elementAt(i)).getQ() < ((Node)scheduled.elementAt(i2)).getQ() ){
           c = ((Node)scheduled.elementAt(i));
      }
    }
  return c;
}

 /*public int indexc (Vector crit, Vector scheduled) {

   int i1 = ((Integer)crit.firstElement()).intValue();
   int i2 = ((Integer)crit.elementAt(2)).intValue();
   int indexc = 0;
   for (int i = i1; i < i2; i++) {

      if ( ((Node)scheduled.elementAt(i)).getQ() < ((Node)scheduled.elementAt(i2)).getQ() ){
           indexc = i;
      }
    }

  return indexc;
}*/

public int retrieveQP (Vector crit, Vector scheduled) {
  int i2 = ((Integer)crit.elementAt(1)).intValue();
```

```
  int qp = ((Node)scheduled.elementAt(i2)).getProc_time();
  return qp;

}

public int sygmaJ(Node c, Vector crit, Vector scheduled) {

  int i2 = ((Integer)crit.lastElement()).intValue();
  int sum = 0;

  for (int i = scheduled.indexOf(c) +1; i <= i2; i++) {
    sum = sum + ((Node)scheduled.elementAt(i)).getProc_time();
  }

  return sum;
}

public int sygmaJ(int indexc, Vector crit, Vector scheduled) {

  int i2 = ((Integer)crit.lastElement()).intValue();
  int sum = 0;

  for (int i = indexc +1; i <= i2; i++) {
    sum = sum + ((Node)scheduled.elementAt(i)).getProc_time();
  }

  return sum;
}

public int minRJ (Node c, Vector crit, Vector scheduled) {

  int i2 = ((Integer)crit.lastElement()).intValue();

  int minrel = Integer.MAX_VALUE;
  for (int i = scheduled.indexOf(c) +1; i <= i2; i++){
    if ( ((Node)scheduled.elementAt(i)).getRel_time() < minrel )
      minrel = ((Node)scheduled.elementAt(i)).getRel_time();
  }

  return minrel;
}

public int minRJ (int indexc, Vector crit, Vector scheduled) {

  int i2 = ((Integer)crit.lastElement()).intValue();

  int minrel = Integer.MAX_VALUE;
  for (int i = indexc +1; i <= i2; i++){
    if ( ((Node)scheduled.elementAt(i)).getRel_time() < minrel )
      minrel = ((Node)scheduled.elementAt(i)).getRel_time();
  }

  return minrel;
}

public int minQJ (Node c, Vector crit, Vector scheduled) {

  int i2 = ((Integer)crit.lastElement()).intValue();

  int minQ = Integer.MAX_VALUE;
  for (int i = scheduled.indexOf(c) +1; i <= i2; i++){
    if ( ((Node)scheduled.elementAt(i)).getQ() < minQ )
      minQ = ((Node)scheduled.elementAt(i)).getQ();
  }

  return minQ;
}

public int minQJ (int indexc, Vector crit, Vector scheduled) {

  int i2 = ((Integer)crit.lastElement()).intValue();

  int minQ = Integer.MAX_VALUE;
  for (int i = indexc +1; i <= i2; i++){
    if ( ((Node)scheduled.elementAt(i)).getQ() < minQ )
      minQ = ((Node)scheduled.elementAt(i)).getQ();
```

```
  }

  return minQ;
}

public int fonctionH (Node c, Vector crit, Vector scheduled){

 return minRJ(c, crit, scheduled) + sygmaJ(c, crit, scheduled) + minQJ(c, crit, scheduled);

}

public int fonctionH (int indexc, Vector crit, Vector scheduled){

 return minRJ(indexc, crit, scheduled) + sygmaJ(indexc, crit, scheduled)
            + minQJ(indexc, crit, scheduled);

}

public TreeNode searchv (Vector V) {
  TreeNode v = (TreeNode)V.firstElement() ;
  for (int i = 0; i < V.size(); i++) {
    if (v.getFi() < ((TreeNode)V.elementAt(i)).getFi() ) {
      v = (TreeNode)V.elementAt(i);
    }
  }
  return v;
}

public TreeNode carlier (Vector input) {

  //Initialization
  Vector tree = new Vector();
  TreeNode v = null;

  // schedule input
  input = schrage(input);
  //System.out.println ("Schedule size "+input.size() );
  TreeNode solution = new TreeNode(input);
  int fsup = computeCmax(input);
  solution.setFo(fsup);
  //probleme si c est null?
  Vector critp = crit_path(input);
  Node c = findc(critp, input);

  int indexc = 0;
  int fo = 0;
  int f= 0;
  if (c == null) return solution;

  // else the algorithm continues

  indexc = input.indexOf(c);
  fo = fonctionH(c, critp, input);
  f  = fo;


  while (c != null) {
    //step1
    Vector scheduled1= input;
    Vector scheduled2 = input;

    ((Node)scheduled1.elementAt(indexc)).setRel_time(Math.max(c.getRel_time(),
    minRJ(indexc, critp, input)+ sygmaJ(indexc, critp,input)));

    ((Node)scheduled2.elementAt(indexc)).setQ(Math.max( c.getQ(),
    retrieveQP(critp, input) + sygmaJ(indexc, critp, input) ));

    int falpha = Math.max(f, fonctionH(indexc-1, critp, scheduled1));
    int fbeta = Math.max(f, fonctionH(indexc-1, critp, scheduled2));

    TreeNode t1 = new TreeNode();
    t1.setScheduled(scheduled1);
    t1.setFi(falpha);
    t1.setFo(fsup);

    TreeNode t2 = new TreeNode();
```

```
    t2.setScheduled(scheduled2);
    t2.setFi(fbeta);
    t2.setFo(fsup);

    if (t1.getFi()<fsup) {
      tree.add(t1);
    }

    if (t2.getFi()<fsup){
      tree.add(t2);
    }

    //step 2

    c = null;
    int fv = 0;

    while ((c==null) && (tree.size()!=0) && (fsup > fv) ) {
      v = searchv(tree);
      tree.removeElement(v);
      input = schrage(v.getScheduled());
      critp= crit_path(input);
      c = findc(critp, input);
      indexc = input.indexOf(c);
      fv = Math.max(fv, fonctionH(indexc, critp, input));
      if (fsup > fv ) {
        fsup = fv;
        solution.setScheduled(input);
        solution.setFo(fv);
      }
    }
  }
return solution;
}

public int computeCmax (Vector m) {
  int cmax = 0;
  if (m.size() > 0) {
    cmax = ((Node)m.elementAt(0)).getProc_time() + ((Node)m.elementAt(0)).getRel_time() ;
    for (int i = 1; i < m.size(); i++) {
      cmax = Math.max(cmax, ((Node)m.elementAt(i)).getRel_time()) + ((Node)m.elementAt(i)).getProc_time();
    }
  } else {
    System.out.println("m est vide");
  }
  //System.out.println ("La valeur de cmax est "+cmax);
  return cmax;
}

public int carlierFo (Vector m) {
  TreeNode t = carlier(m);
  m = t.getScheduled();
  int fo = t.getFo();
  return fo;
}

public int maxCarlier (Vector mlist) {
  int index = 0;
  int max = 0; //carlierFo(((Vector)mlist.elementAt(0)));
  for (int i = 0; i < mlist.size(); i++) {
    int tmp = carlierFo(((Vector)mlist.elementAt(i)));
    if (tmp > max) {
      index  = i;
      max =  tmp;
    }
  }
  return index;
}


//Apply Carlier to each element of m, get the index of highest value of Carlier obtained
/*public int maxCarlier (Vector m) {
  TreeNode first = carlier(((Vector)m.firstElement()));
  int max = first.getFo();
  int index = 0;
  for (int i=0; i < m.size(); i++) {
```

```
  TreeNode t = carlier(((Vector)m.elementAt(i)));
    if (max < t.getFo()) {
    max = t.getFo();
    index = i;
    }
  }
return index;
}*/

public DirectedGraphImpl createGraph (Vector plist) {

  DirectedGraphImpl graph  = new DirectedGraphImpl();
  VertexImpl source = new VertexImpl();
  source.setString("source");
  VertexImpl sink = new VertexImpl();
  sink.setString("sink");

  try {
     graph.add(source);}
     catch (Exception ex){System.err.println("erreur d'insertion de source");}

  try {
     graph.add(sink);}
     catch (Exception ex){System.err.println("erreur d'insertion de sink");}


  for (int i=0; i < plist.size(); i++) {
    VertexImpl prev = source;

    for (int j = 0; j < ((Vector)plist.elementAt(i)).size(); j++) {
      VertexImpl v = new VertexImpl((Node)((Vector)plist.elementAt(i)).elementAt(j));
      v.setString("v"+ i+j);
      double weight;
      if (j ==0) {
        weight = 0;
      } else {
        weight = ((Node)((Vector)plist.elementAt(i)).elementAt(j-1)).getProc_time();
      }
      DirectedWeightedEdgeImpl e = new DirectedWeightedEdgeImpl(prev, v, weight);
      try {
      graph.addEdge(e);}
      catch (Exception ex){System.err.println("erreur d'insertion de l'arc interne");}
      prev = v;
    }
    DirectedWeightedEdgeImpl e = new DirectedWeightedEdgeImpl(prev, sink, ((Node)prev.getObject()).getProc_time() );
    try {
    graph.addEdge(e);
    } catch (Exception ex){System.err.println("erreur d'insertion de l'arc vers le sink");}
  }

  return graph;
}

public DirectedGraphImpl createGraph2 (Vector plist, int[] transition) {

  DirectedGraphImpl graph  = new DirectedGraphImpl();
  VertexImpl source = new VertexImpl();
  source.setString("source");
  VertexImpl sink = new VertexImpl();
  sink.setString("sink");

  try {
     graph.add(source);}
     catch (Exception ex){System.err.println("erreur d'insertion de source");}

  try {
     graph.add(sink);}
     catch (Exception ex){System.err.println("erreur d'insertion de sink");}


  for (int i=0; i < plist.size(); i++) {
    VertexImpl prev = source;

    for (int j = 0; j < ((Vector)plist.elementAt(i)).size(); j++) {
      VertexImpl v = new VertexImpl((Node)((Vector)plist.elementAt(i)).elementAt(j));
      v.setString("v"+ i+j);
```

```
      double weight;
      if (j == 0) {
        weight = transition[i];
      } else {
        weight = ((Node)((Vector)plist.elementAt(i)).elementAt(j-1)).getProc_time();
      }
      DirectedWeightedEdgeImpl e = new DirectedWeightedEdgeImpl(prev, v, weight);
      try {
      graph.addEdge(e);}
      catch (Exception ex){System.err.println("erreur d'insertion de l'arc interne");}
      prev = v;
    }
    DirectedWeightedEdgeImpl e = new DirectedWeightedEdgeImpl(prev, sink, ((Node)prev.getObject()).getProc_time() );
    try {
    graph.addEdge(e);
    } catch (Exception ex){System.err.println("erreur d'insertion de l'arc vers le sink");}
  }

  return graph;
}


public VertexImpl getsink(DirectedGraph g) {

  VertexImpl sink = new VertexImpl();
  Iterator iter = g.getVerticesIterator();
  while(iter.hasNext()){
    VertexImpl v = (VertexImpl)iter.next();
    if ( g.getOutgoingEdges(v).size() == 0 ) {
    sink = v;
    }
  }
//System.err.println("le sink c'est "+sink.toString());
return sink;

}

public double critpath (DirectedGraph g, Vertex v) {

double crit = 0;

// pour un sommet convergent

if (g.getIncomingEdges(v).size() > 1 ) {
  Iterator i = g.getIncomingEdges(v).iterator();
  DirectedWeightedEdgeImpl ed = (DirectedWeightedEdgeImpl)i.next();
  crit = critpath (g, ed.getSource()) + ed.getWeight();

  while (i.hasNext()) {
    DirectedWeightedEdgeImpl e = (DirectedWeightedEdgeImpl)i.next();
    double temp = critpath (g, e.getSource()) + e.getWeight();
    if ( temp > crit ) {
     crit = temp ;
    }
  }

}
  else {
    //pour un sommet ayant un seul antécédent
    if ( g.getIncomingEdges(v).size() == 1 ) {
      Iterator i = g.getIncomingEdges(v).iterator();

      while (i.hasNext()) {
        DirectedWeightedEdgeImpl edge = (DirectedWeightedEdgeImpl)i.next();
        crit = critpath (g, edge.getSource()) + edge.getWeight();
      }
    }
      else {
      //pour le source
      crit = 0;
    }
  }

return crit;

}
```

```
public VertexImpl findVertex (DirectedGraph g, Node n) {
  VertexImpl v = null;
  Iterator iter = g.getVerticesIterator();
  while(iter.hasNext()){
    v = (VertexImpl) iter.next();
    if (v.getObject() != null && v.getObject().equals(n))
      return v;
  }
  return v;
}

public DirectedEdge findEdge (DirectedGraph g, Node n1, Node n2) {
VertexImpl v1 = findVertex (g, n1);
VertexImpl v2 = findVertex (g, n2);
DirectedEdge e = g.getEdge(v1,v2);
return e;
}


public DirectedGraph addMachine (DirectedGraph g, Vector m) {
  DirectedGraph graph = g;
  for (int i=1; i < m.size(); i++) {
  VertexImpl prev = findVertex(graph,(Node)m.elementAt(i-1));
  VertexImpl v = findVertex(graph,(Node)m.elementAt(i));
  double weight = ((Node)m.elementAt(i-1)).getProc_time();
  DirectedWeightedEdgeImpl e = null;
  if (prev != null && v != null)  {
    e = new DirectedWeightedEdgeImpl(prev, v, weight);
  } else {
     System.err.println("zut ");
     System.exit(0);
  }
  try {
      graph.addEdge(e);}
      catch (Exception ex){System.err.println("erreur d'insertion");}
  }
return graph;
}

public DirectedGraph plusEdge (DirectedGraph g, Node n1, Node n2) {
  VertexImpl prev = findVertex(g, n1);
  VertexImpl v = findVertex(g, n2);
  double weight = n1.getProc_time();
  DirectedWeightedEdgeImpl e = null;
  if (prev != null && v != null)  {
    e = new DirectedWeightedEdgeImpl(prev, v, weight);
  } else {
     System.err.println("zut ");
     System.exit(0);
  }
  try {
      g.addEdge(e);}
      catch (Exception ex){System.err.println("erreur d'insertion");}

  return g;
}

public DirectedGraph plusMachine (DirectedGraph g, Vector m) {
  for (int i=1; i < m.size(); i++) {
    plusEdge(g,(Node)m.elementAt(i-1), (Node)m.elementAt(i));
  }
  return g;
}

public DirectedGraph delMachine (DirectedGraph g, Vector m) {
  for (int i=0; i < m.size()-1; i++) {
    try {
      g.removeEdge(findEdge(g,(Node)m.elementAt(i), (Node)m.elementAt(i+1)));}
    catch (Exception ex){System.err.println("erreur de suppression d'arc");}
  }
  return g;
}

public DirectedGraph reoptimStep (DirectedGraph g, Vector schedmachines){
  DirectedGraph graph = g;
```

```
 return graph;
}

public void updateData (DirectedGraph g, Vector mlist, Vector plist) {
 for (int i = 0; i < mlist.size(); i++) {
  for (int j = 0; j < ((Vector)mlist.elementAt(i)).size(); j ++) {
   updateRel(g,((Node)((Vector)mlist.elementAt(i)).elementAt(j)));
   updateQ(g,((Node)((Vector)mlist.elementAt(i)).elementAt(j)), plist);
  }
 }
}

public void updateRel (DirectedGraph g, Node n) {
 n.setRel_time(((int)critpath(g, findVertex(g, n))));
}

public void updateQ (DirectedGraph g, Node n, Vector plist) {
 double q = critpath(g, findVertex(g,((Node)((Vector)plist.elementAt(n.getProduct()-1)).lastElement())))
 + ((Node)((Vector)plist.elementAt(n.getProduct()-1)).lastElement()).getProc_time()
  - critpath(g, findVertex(g,n)) - n.getProc_time();
 n.setQ((int)q);
}

public void printmlist (Vector mlist) {
 for (int i = 0; i < mlist.size(); i++) {
  for (int j = 0; j < ((Vector)mlist.elementAt(i)).size(); j ++) {
  ((Node)((Vector)mlist.elementAt(i)).elementAt(j)).print();
  }
 }
}

public double shiftB (DirectedGraph g, Vector mlist, Vector plist) {
 int index = mlist.size();
 Vector mlist0 = new Vector();
 double cmaxg;
 for (int j = 0; j < index; j++) {
    int i = maxCarlier(mlist);
    TreeNode t = carlier(((Vector)mlist.elementAt(i)));
    mlist0.addElement(t.getScheduled());
    g = addMachine(g, t.getScheduled());
    mlist.removeElementAt(i);
    updateData(g, mlist, plist);
 }
 updateData(g, mlist0, plist);
 cmaxg = critpath(g, getsink(g));
 System.err.println("Le Cmax de l'Ordonnancement vaut "+cmaxg);
 return cmaxg;
}

public double ShiftingBottleneck (Window win) {

 double cmax;
 DirectedGraphImpl g = createGraph(win.plist);
 Vector mlist = win.machines();
 updateData(g, mlist, win.plist);
 cmax = shiftB(g, mlist, (Vector)win.plist);
 //System.err.println("Le Cmax de l'Ordonnancement vaut "+cmax);
 return cmax;
}

 public Vector ShiftingBottleneck2 (Window win) {

 double cmaxg;
 Vector mlist0 = new Vector();;
 DirectedGraphImpl g = createGraph(win.plist);
 Vector mlist = win.machines();
 updateData(g, mlist, win.plist);
 mlist0 = shiftB2(g, mlist, (Vector)win.plist);
 cmaxg = critpath(g, getsink(g));
 System.err.println("Le Cmax de l'Ordonnancement vaut "+cmaxg);
 return mlist0;
}

public Vector shiftB2 (DirectedGraph g, Vector mlist, Vector plist) {
 int index = mlist.size();
 Vector mlist0 = new Vector();
```

```
  double cmaxg;
  for (int j = 0; j < index; j++) {
     int i = maxCarlier(mlist);
     TreeNode t = carlier(((Vector)mlist.elementAt(i)));
     mlist0.addElement(t.getScheduled());
     g = addMachine(g, t.getScheduled());
     mlist.removeElementAt(i);
     updateData(g, mlist, plist);
  }
  updateData(g, mlist0, plist);
  return mlist0;
}

public void ajustMachine (DirectedGraph g, Vector mlist1, Vector m){
  Node n2 = (Node)m.firstElement();
  int indexin1 = machineIndex(mlist1, n2);
  if (indexin1 != -1){
    Node n1 = (Node)((Vector)mlist1.elementAt(indexin1)).lastElement();
    plusEdge(g, n1, n2);
  }
}

public Vector shiftB2b (DirectedGraph g, Vector mlist1, Vector mlist2, Vector plist2) {
  int index = mlist2.size();
  Vector mlist0 = new Vector();
  double cmaxg;
  for (int j = 0; j < index; j++) {
     int i = maxCarlier(mlist2);
     TreeNode t = carlier(((Vector)mlist2.elementAt(i)));
     mlist0.addElement(t.getScheduled());
     g = addMachine(g, t.getScheduled());
     ajustMachine(g, mlist1, t.getScheduled());
     mlist2.removeElementAt(i);
     updateData(g, mlist2, plist2);
  }
  updateData(g, mlist0, plist2);
  return mlist0;
}

public int getIndexNode (Vector m, Node n) {
  int index = 0;
  for (int i =0; i < m.size(); i++) {
    int mac = ((Node)m.elementAt(i)).getMachine() ;
    int prod = ((Node)m.elementAt(i)).getProduct();
      if (mac == n.getMachine() && prod == n.getProduct()) {
        index = i;
      }
  }
  return index;
}

public int getcompletion (Vector m, Node n) {
  int index = getIndexNode(m,n);
  int completion = ((Node)m.firstElement()).getRel_time() + ((Node)m.firstElement()).getProc_time();
  if ( index !=0) {
    for (int i= 1; i <= index; i++) {
      completion = Math.max(completion, ((Node)m.elementAt(i)).getRel_time())
            + ((Node)m.elementAt(i)).getProc_time();
    }
  }
  return completion;
}

public Vector transtion (Vector plist) {
  Vector transition = new Vector();
  for (int i = 0; i < plist.size(); i ++) {
    Node n = ((Node)((Vector)plist.elementAt(i)).lastElement());
    transition.addElement(n);
  }
  return transition;
}

public int machineIndex (Vector mlist, Node n) {
  int index = -1;
   for (int i =0; i < mlist.size(); i++) {
    int mac = ((Node)((Vector)mlist.elementAt(i)).firstElement()).getMachine() ;
```

```
      if (mac == n.getMachine()) {
        index = i;
      }
   }
   return index;

}

 public int machineIndex2 (Vector mlist, int m) {
   int index = -1;
    for (int i =0; i < mlist.size(); i++) {
     int mac = ((Node)((Vector)mlist.elementAt(i)).firstElement()).getMachine() ;
       if (mac == m) {
         index = i;
       }
   }
   return index;

}

public int[] transitionvalues (Vector transition, Vector mlist) {
  int size = transition.size();
  int[] table = new int[size];
  for (int i=0; i < transition.size(); i++) {
    Node n = (Node)transition.elementAt(i);
    int mindex = machineIndex(mlist, n);
    if (mindex != -1){
      int comp = getcompletion( (Vector)mlist.elementAt(mindex), n);
      table[i] = comp;
    } else {
      System.err.println("Grosse erreur in transitionvalues");
    }
  }
  return table;
}

public Vector removeNull (Vector mlist) {

  Vector mlist0 = new Vector();
  for (int i = 0; i < mlist.size(); i++) {
    if (((Vector)mlist.elementAt(i)).size() != 0) mlist0.addElement((Vector)mlist.elementAt(i));
  }
  return mlist0;
}

public double decompositonSB10_2 (Window win) {
  double cmax =0;
  Window w1 = win.getSubWindow(0,5);
  Vector transition1 = transtion (w1.plist);
  Window w2 = win.getSubWindow(6,9);

  //gestion de la 1ère fenêtre
  DirectedGraphImpl g1 = createGraph(w1.plist);
  Vector mlist1 = w1.machines();
  mlist1 = removeNull(mlist1);
  updateData(g1, mlist1, w1.plist);
  Vector mlist1end = shiftB2(g1, mlist1, w1.plist);
  int[] table1 = new int[transition1.size()];
  table1 = transitionvalues(transition1, mlist1end);

  //gestion de la 2e fenêtre
  DirectedGraphImpl g2 = createGraph2(w2.plist, table1);
  Vector mlist2 = w2.machines();
  mlist2 = removeNull(mlist2);
  updateData(g2, mlist2, w2.plist);
  Vector mlist2end = shiftB2(g2, mlist2, w2.plist);

  //raccordement des fenêtres
  ajustWindows(g2, mlist1end, mlist2end);

  //mise à jour après raccordement
  updateData(g2, mlist2end, w2.plist);

  cmax = critpath(g2, getsink(g2));

  /*printmlist(mlist1end);
```

```
  printmlist(mlist2end);*/

  System.err.println("Le Cmax avec décomposition en 2 : "+cmax);
  return cmax;
 }


 public double decompositonSB10_3 (Window win) {
  double cmax =0;
  Window w1 = win.getSubWindow(0,2);
  Vector transition1 = transtion (w1.plist);
  Window w2 = win.getSubWindow(3,5);
  Vector transition2 = transtion (w2.plist);
  Window w3 = win.getSubWindow(6,9);

  //gestion de la 1ère fenêtre
  DirectedGraphImpl g1 = createGraph(w1.plist);
  Vector mlist1 = w1.machines();
  mlist1 = removeNull(mlist1);
  updateData(g1, mlist1, w1.plist);
  Vector mlist1end = shiftB2(g1, mlist1, w1.plist);
  int[] table1 = new int[transition1.size()];
  table1 = transitionvalues(transition1, mlist1end);

  //gestion de la 2e fenêtre
  DirectedGraphImpl g2 = createGraph2(w2.plist, table1);
  Vector mlist2 = w2.machines();
  mlist2 = removeNull(mlist2);
  updateData(g2, mlist2, w2.plist);
  Vector mlist2end = shiftB2(g2, mlist2, w2.plist);

  //raccordement des fenêtres
  ajustWindows(g2, mlist1end, mlist2end);

  //mise à jour après raccordement
  updateData(g2, mlist2end, w2.plist);

  int [] table2 = new int[transition2.size()];
  table2 = transitionvalues(transition2, mlist2end);

  //gestion de la 3e fenêtre
  DirectedGraphImpl g3 = createGraph2(w3.plist, table2);
  Vector mlist3 = w3.machines();
  mlist3 = removeNull(mlist3);
  updateData(g3, mlist3, w3.plist);
  Vector mlist3end = shiftB2(g3, mlist3, w3.plist);

  //raccordement des fenêtres
  ajustWindows(g3, mlist2end, mlist3end);

  //mise à jour après raccordement
  updateData(g3, mlist3end, w3.plist);

  cmax = critpath(g3, getsink(g3));

  /*printmlist(mlist1end);
  printmlist(mlist2end);
  printmlist(mlist3end);*/

  System.err.println("Cmax avec décomposition en 3 : "+cmax);

  return cmax;
 }


 public double decompositonSB10_5 (Window win) {
  double cmax =0;
  Window w1 = win.getSubWindow(0,1);
  Vector transition1 = transtion (w1.plist);
  Window w2 = win.getSubWindow(2,3);
  Vector transition2 = transtion (w2.plist);
  Window w3 = win.getSubWindow(4,5);
  Vector transition3 = transtion (w3.plist);
  Window w4 = win.getSubWindow(6,7);
  Vector transition4 = transtion (w4.plist);
  Window w5 = win.getSubWindow(8,9);
```

```
//gestion de la 1ère fenêtre
DirectedGraphImpl g1 = createGraph(w1.plist);
Vector mlist1 = w1.machines();
mlist1 = removeNull(mlist1);
updateData(g1, mlist1, w1.plist);
Vector mlist1end = shiftB2(g1, mlist1, w1.plist);
int[] table1 = new int[transition1.size()];
table1 = transitionvalues(transition1, mlist1end);

//gestion de la 2e fenêtre
DirectedGraphImpl g2 = createGraph2(w2.plist, table1);
Vector mlist2 = w2.machines();
mlist2 = removeNull(mlist2);
updateData(g2, mlist2, w2.plist);
Vector mlist2end = shiftB2(g2, mlist2, w2.plist);

//raccordement des fenêtres
ajustWindows(g2, mlist1end, mlist2end);

//mise à jour après raccordement
updateData(g2, mlist2end, w2.plist);

int [] table2 = new int[transition2.size()];
table2 = transitionvalues(transition2, mlist2end);

//gestion de la 3e fenêtre
DirectedGraphImpl g3 = createGraph2(w3.plist, table2);
Vector mlist3 = w3.machines();
mlist3 = removeNull(mlist3);
updateData(g3, mlist3, w3.plist);
Vector mlist3end = shiftB2(g3, mlist3, w3.plist);

//raccordement des fenêtres
ajustWindows(g3, mlist2end, mlist3end);

//mise à jour après raccordement
updateData(g3, mlist3end, w3.plist);

int [] table3 = new int[transition3.size()];
table3 = transitionvalues(transition3, mlist3end);

 //gestion de la 4e fenêtre
DirectedGraphImpl g4 = createGraph2(w4.plist, table3);
Vector mlist4 = w4.machines();
mlist4 = removeNull(mlist4);
updateData(g4, mlist4, w4.plist);
Vector mlist4end = shiftB2(g4, mlist4, w4.plist);

//raccordement des fenêtres
ajustWindows(g4, mlist3end, mlist4end);

//mise à jour après raccordement
updateData(g4, mlist4end, w4.plist);

int [] table4 = new int[transition4.size()];
table4 = transitionvalues(transition4, mlist4end);

//gestion de la 5e fenêtre
DirectedGraphImpl g5 = createGraph2(w5.plist, table4);
Vector mlist5 = w5.machines();
mlist5 = removeNull(mlist5);
updateData(g5, mlist5, w5.plist);
Vector mlist5end = shiftB2(g5, mlist5, w5.plist);

//raccordement des fenêtres
ajustWindows(g5, mlist4end, mlist5end);

//mise à jour après raccordement
updateData(g5, mlist5end, w5.plist);

cmax = critpath(g5, getsink(g5));

/*printmlist(mlist1end);
printmlist(mlist2end);
printmlist(mlist3end);
```

```
  printmlist(mlist4end);
  printmlist(mlist5end);*/

  System.err.println("Cmax avec décomposition en 5 : "+cmax);

  return cmax;
 }

 public double decompositonSB15 (Window win) {
  double cmax =0;
  Window w1 = win.getSubWindow(0,4);
  Vector transition1 = transtion (w1.plist);
  Window w2 = win.getSubWindow(5,9);
  Vector transition2 = transtion (w2.plist);
  Window w3 = win.getSubWindow(9,14);

  //gestion de la 1ère fenêtre
  DirectedGraphImpl g1 = createGraph(w1.plist);
  Vector mlist1 = w1.machines();
  mlist1 = removeNull(mlist1);
  updateData(g1, mlist1, w1.plist);
  Vector mlist1end = shiftB2(g1, mlist1, w1.plist);
  int[] table1 = new int[transition1.size()];
  table1 = transitionvalues(transition1, mlist1end);

  //gestion de la 2e fenêtre
  DirectedGraphImpl g2 = createGraph2(w2.plist, table1);
  Vector mlist2 = w2.machines();
  mlist2 = removeNull(mlist2);
  updateData(g2, mlist2, w2.plist);
  Vector mlist2end = shiftB2(g2, mlist2, w2.plist);

  //raccordement des fenêtres
  ajustWindows(g2, mlist1end, mlist2end);

  //mise à jour après raccordement
  updateData(g2, mlist2end, w2.plist);

  int [] table2 = new int[transition2.size()];
  table2 = transitionvalues(transition2, mlist2end);

  //gestion de la 3e fenêtre
  DirectedGraphImpl g3 = createGraph2(w3.plist, table2);
  Vector mlist3 = w3.machines();
  mlist3 = removeNull(mlist3);
  updateData(g3, mlist3, w3.plist);
  Vector mlist3end = shiftB2(g3, mlist3, w3.plist);

  //raccordement des fenêtres
  ajustWindows(g3, mlist2end, mlist3end);

  //mise à jour après raccordement
  updateData(g3, mlist3end, w3.plist);

  cmax = critpath(g3, getsink(g3));

  System.err.println("Le Cmax avec décomposition "+cmax);

  return cmax;
 }

 public double decompositonSB5_2 (Window win) {
  double cmax =0;
  Window w1 = win.getSubWindow(0,2);
  Vector transition1 = transtion (w1.plist);
  Window w2 = win.getSubWindow(3,4);

  //gestion de la 1ère fenêtre
  DirectedGraphImpl g1 = createGraph(w1.plist);
  Vector mlist1 = w1.machines();
  mlist1 = removeNull(mlist1);
  updateData(g1, mlist1, w1.plist);
  Vector mlist1end = shiftB2(g1, mlist1, w1.plist);
  int[] table1 = new int[transition1.size()];
  table1 = transitionvalues(transition1, mlist1end);
```

```
  //gestion de la 2e fenêtre
  DirectedGraphImpl g2 = createGraph2(w2.plist, table1);
  Vector mlist2 = w2.machines();
  mlist2 = removeNull(mlist2);
  updateData(g2, mlist2, w2.plist);
  Vector mlist2end = shiftB2(g2, mlist2, w2.plist);

  //raccordement des fenêtres
  ajustWindows(g2, mlist1end, mlist2end);

  //mise à jour après raccordement
  updateData(g2, mlist2end, w2.plist);

  cmax = critpath(g2, getsink(g2));

  /*printmlist(mlist1end);
  printmlist(mlist2end);*/

  System.err.println("Le Cmax avec décomposition en 2 : "+cmax);
  return cmax;
 }

 public double decompositonSB5_3 (Window win) {
  double cmax =0;
  Window w1 = win.getSubWindow(0,1);
  Vector transition1 = transtion (w1.plist);
  Window w2 = win.getSubWindow(2,3);
  Vector transition2 = transtion (w2.plist);
  Window w3 = win.getSubWindow(4,4);

  //gestion de la 1ère fenêtre
  DirectedGraphImpl g1 = createGraph(w1.plist);
  Vector mlist1 = w1.machines();
  mlist1 = removeNull(mlist1);
  updateData(g1, mlist1, w1.plist);
  Vector mlist1end = shiftB2(g1, mlist1, w1.plist);
  int[] table1 = new int[transition1.size()];
  table1 = transitionvalues(transition1, mlist1end);

  //gestion de la 2e fenêtre
  DirectedGraphImpl g2 = createGraph2(w2.plist, table1);
  Vector mlist2 = w2.machines();
  mlist2 = removeNull(mlist2);
  updateData(g2, mlist2, w2.plist);
  Vector mlist2end = shiftB2(g2, mlist2, w2.plist);

  //raccordement des fenêtres
  ajustWindows(g2, mlist1end, mlist2end);

  //mise à jour après raccordement
  updateData(g2, mlist2end, w2.plist);

  int [] table2 = new int[transition2.size()];
  table2 = transitionvalues(transition2, mlist2end);

  //gestion de la 3e fenêtre
  DirectedGraphImpl g3 = createGraph2(w3.plist, table2);
  Vector mlist3 = w3.machines();
  mlist3 = removeNull(mlist3);
  updateData(g3, mlist3, w3.plist);
  Vector mlist3end = shiftB2(g3, mlist3, w3.plist);

  //raccordement des fenêtres
  ajustWindows(g3, mlist2end, mlist3end);

  //mise à jour après raccordement
  updateData(g3, mlist3end, w3.plist);

  /*printmlist(mlist1end);
  printmlist(mlist2end);
  printmlist(mlist3end);*/

  cmax = critpath(g3, getsink(g3));

  System.err.println("Le Cmax avec décomposition en 3 : "+cmax);
```

```
  return cmax;
}


public double decompositonSB5b (Window win) {
  double cmax =0;

  DirectedGraphImpl g = createGraph(win.plist);

  Window w1 = win.getSubWindow(0,2);
  Vector transition1 = transtion (w1.plist);
  Window w2 = win.getSubWindow(3,4);

  //gestion de la 1ère fenêtre

  Vector mlist1 = w1.machines();
  mlist1 = removeNull(mlist1);
  updateData(g, mlist1, w1.plist);
  Vector mlist1end = shiftB2(g, mlist1, w1.plist);
  int[] table1 = new int[transition1.size()];
  table1 = transitionvalues(transition1, mlist1end);

  //gestion de la 2e fenêtre

  Vector mlist2 = w2.machines();
  mlist2 = removeNull(mlist2);
  updateData(g, mlist2, w2.plist);
  Vector mlist2end = shiftB2b(g, mlist1end, mlist2, w2.plist);

  //mise à jour après raccordement
  updateData(g, mlist2end, w2.plist);

  cmax = critpath(g, getsink(g));

  System.err.println("Le Cmax avec décompositionb "+cmax);
  return cmax;
}

public double decompositonSB10b (Window win) {
  double cmax =0;

  DirectedGraphImpl g = createGraph(win.plist);

  Window w1 = win.getSubWindow(0,2);
  Vector transition1 = transtion (w1.plist);
  Window w2 = win.getSubWindow(3,5);
  Vector transition2 = transtion (w2.plist);
  Window w3 = win.getSubWindow(6,9);

  //gestion de la 1ère fenêtre

  Vector mlist1 = w1.machines();
  mlist1 = removeNull(mlist1);
  updateData(g, mlist1, w1.plist);
  Vector mlist1end = shiftB2(g, mlist1, w1.plist);
  int[] table1 = new int[transition1.size()];
  table1 = transitionvalues(transition1, mlist1end);

  //gestion de la 2e fenêtre

  Vector mlist2 = w2.machines();
  mlist2 = removeNull(mlist2);
  updateData(g, mlist2, w2.plist);
  Vector mlist2end = shiftB2b(g, mlist1end, mlist2, w2.plist);

  //mise à jour après raccordement
  updateData(g, mlist2end, w2.plist);

  int[] table2 = new int[transition2.size()];
  table2 = transitionvalues(transition2, mlist2end);

  //gestion de la 3e fenêtre

  Vector mlist3 = w3.machines();
  mlist3 = removeNull(mlist3);
  updateData(g, mlist3, w3.plist);
```

```
  Vector mlist3end = shiftB2b(g, mlist2end, mlist3, w3.plist);

  //mise à jour après raccordement
  updateData(g, mlist3end, w3.plist);

  cmax = critpath(g, getsink(g));

  System.err.println("Le Cmax avec décompositionb "+cmax);
  return cmax;
}

public double decompositonSB15b (Window win) {
  double cmax =0;

  DirectedGraphImpl g = createGraph(win.plist);

  Window w1 = win.getSubWindow(0,4);
  Vector transition1 = transtion (w1.plist);
  Window w2 = win.getSubWindow(5,9);
  Vector transition2 = transtion (w2.plist);
  Window w3 = win.getSubWindow(9,14);


  //gestion de la 1ère fenêtre

  Vector mlist1 = w1.machines();
  mlist1 = removeNull(mlist1);
  updateData(g, mlist1, w1.plist);
  Vector mlist1end = shiftB2(g, mlist1, w1.plist);
  int[] table1 = new int[transition1.size()];
  table1 = transitionvalues(transition1, mlist1end);

  //gestion de la 2e fenêtre

  Vector mlist2 = w2.machines();
  mlist2 = removeNull(mlist2);
  updateData(g, mlist2, w2.plist);
  Vector mlist2end = shiftB2b(g, mlist1end, mlist2, w2.plist);

  //mise à jour après raccordement
  updateData(g, mlist2end, w2.plist);

  int[] table2 = new int[transition2.size()];
  table2 = transitionvalues(transition2, mlist2end);

  //gestion de la 4e fenêtre

  Vector mlist3 = w3.machines();
  mlist3 = removeNull(mlist3);
  updateData(g, mlist3, w3.plist);
  Vector mlist3end = shiftB2b(g, mlist2end, mlist3, w3.plist);

  //mise à jour après raccordement
  updateData(g, mlist3end, w3.plist);

  cmax = critpath(g, getsink(g));

  System.err.println("Le Cmax avec décompositionb "+cmax);
  return cmax;
}

public void ajustWindows (DirectedGraph g2, Vector mlist1, Vector mlist2) {

  for (int i = 0; i< mlist2.size(); i++) {
    Node n = (Node)((Vector)mlist2.elementAt(i)).firstElement();
    int indexin1 = machineIndex(mlist1, n);
    if (indexin1 != -1){
      int cmax1 = computeCmax((Vector)mlist1.elementAt(indexin1));
      if (n.getRel_time() < cmax1) {
        VertexImpl v = findVertex(g2, n);
        Iterator iter = g2.getIncomingEdges(v).iterator();
        while (iter.hasNext()) {
          DirectedWeightedEdgeImpl e = (DirectedWeightedEdgeImpl)iter.next();
          double weight = e.getWeight();
          e.setWeight(weight + cmax1 - n.getRel_time());
        }
```

```
      }
     }
    }
   }

/*  public double decompositonSB (Window win) {
    double cmax =0;
    int size = win.getWidth();
    Window w1 = win.getSubWindow(0,1);
    Vector transition1 = transtion (w1.plist);
    Window w2 = win.getSubWindow(2,3);

    //gestion de la 1ère fenêtre
    DirectedGraphImpl g1 = createGraph(w1.plist);
    Vector mlist1 = w1.machines();
    mlist1 = removeNull(mlist1);
    updateData(g1, mlist1, w1.plist);
    Vector mlist1end = shiftB2(g1, mlist1, w1.plist);
    int[] table1 = new int[transition1.size()];
    table1 = transitionvalues(transition1, mlist1end);

    //gestion de la 2e fenêtre
    DirectedGraphImpl g2 = createGraph2(w2.plist, table1);
    Vector mlist2 = w2.machines();
    mlist2 = removeNull(mlist2);
    updateData(g2, mlist2, w2.plist);
    Vector mlist2end = shiftB2(g2, mlist2, w2.plist);

    //raccordement des fenêtres
    ajustWindows(g2, mlist1end, mlist2end);

    //mise à jour après raccordement
    updateData(g2, mlist2end, w2.plist);

    cmax = critpath(g2, getsink(g2));

    System.err.println("Le Cmax avec décomposition "+cmax);
    return cmax;
   }*/

   public double decompositonSB4 (Window win) {
    double cmax =0;
    Window w1 = win.getSubWindow(0,1);
    Vector transition1 = transtion (w1.plist);
    Window w2 = win.getSubWindow(2,3);

    //gestion de la 1ère fenêtre
    DirectedGraphImpl g1 = createGraph(w1.plist);
    Vector mlist1 = w1.machines();
    mlist1 = removeNull(mlist1);
    updateData(g1, mlist1, w1.plist);
    Vector mlist1end = shiftB2(g1, mlist1, w1.plist);
    int[] table1 = new int[transition1.size()];
    table1 = transitionvalues(transition1, mlist1end);

    //gestion de la 2e fenêtre
    DirectedGraphImpl g2 = createGraph2(w2.plist, table1);
    Vector mlist2 = w2.machines();
    mlist2 = removeNull(mlist2);
    updateData(g2, mlist2, w2.plist);
    Vector mlist2end = shiftB2(g2, mlist2, w2.plist);

    //raccordement des fenêtres
    ajustWindows(g2, mlist1end, mlist2end);

    //mise à jour après raccordement
    updateData(g2, mlist2end, w2.plist);

    cmax = critpath(g2, getsink(g2));

    System.err.println("Le Cmax avec décomposition "+cmax);
    return cmax;
   }

   public double decompositonSB4b (Window win) {
    double cmax =0;
```

```
  DirectedGraphImpl g = createGraph(win.plist);

  Window w1 = win.getSubWindow(0,1);
  Vector transition1 = transtion (w1.plist);
  Window w2 = win.getSubWindow(2,3);

  //gestion de la 1ère fenêtre

  Vector mlist1 = w1.machines();
  mlist1 = removeNull(mlist1);
  updateData(g, mlist1, w1.plist);
  Vector mlist1end = shiftB2(g, mlist1, w1.plist);
  int[] table1 = new int[transition1.size()];
  table1 = transitionvalues(transition1, mlist1end);

  //gestion de la 2e fenêtre

  Vector mlist2 = w2.machines();
  mlist2 = removeNull(mlist2);
  updateData(g, mlist2, w2.plist);
  Vector mlist2end = shiftB2b(g, mlist1end, mlist2, w2.plist);

  //mise à jour après raccordement
  updateData(g, mlist2end, w2.plist);

  cmax = critpath(g, getsink(g));

  System.err.println("Le Cmax avec décomposition "+cmax);
  return cmax;
}

public Vector extractFixed (int t, Vector m) {
  Vector m2 = new Vector();
  int index = startFrom(t, m);
  if (index != 0 && index != -1) {
     for (int i=0; i < index; i++) {
      m2.addElement((Node)m.elementAt(i));
    }
  }
  return m2;
}

public int startFrom (int t, Vector m) {

  int index = -1;
  int repere =100;
  int j = 0;
  while ( repere!=0 && j< m.size()) {
    Node n = (Node)m.elementAt(j);
    if (n.getRel_time() >= t) {
      index = j;
      repere = 0;
    }
    else {
      if (getcompletion(m,n) >= t) {
        index = j+1;
        repere = 0;
      }
    }
    j++;
  }
  return index;
}

public Vector extractfromMachine (int t, Vector m) {
  Vector m2 = new Vector();
  int index = startFrom(t, m);
  if (index != -1) {
    for (int i=index; i < m.size(); i++) {
      m2.addElement((Node)m.elementAt(i));
    }
  }
  return m2;
}
```

```
 public Vector startRightmachine (int x, Vector m) {
  Vector startrightmachine = new Vector();
  startrightmachine.addElement(m.elementAt(m.size()-x-1));
  startrightmachine.addElement(m.elementAt(m.size()-x));
  return startrightmachine;
}

public Vector startRight (int x, Vector mlist) {
Vector start = new Vector ();
  for (int i = 0; i < mlist.size(); i++) {
    Vector temp = startRightmachine(x, (Vector)mlist.elementAt(i));
    start.addElement(temp);
  }
  return start;
}

public Vector startLeftmachine (int t, Vector m) {
  Vector startleftmachine = new Vector();
  int index = startFrom (t, m);

  //condition if a revoir
  if (index != 0 && index != -1) {
    startleftmachine.addElement(m.elementAt(index-1));
    startleftmachine.addElement(m.elementAt(index));
  }
  return startleftmachine;
}

public Vector startLeft (int t, Vector mlist) {
  Vector startleft = new Vector ();
  for (int i = 0; i < mlist.size(); i++) {
    Vector temp = startLeftmachine(t, (Vector)mlist.elementAt(i));
    startleft.addElement(temp);
  }
  return startleft;
}



public Vector xextractfromMachine (int t, Vector m, int xleft) {
  Vector m2 = new Vector();
  int index = startFrom(t, m);
  if (index != -1 && index < m.size()-xleft) {
    for (int i=index; i < m.size()-xleft; i++) {
      m2.addElement((Node)m.elementAt(i));
    }
  }
  return m2;

}

public Vector xlefTransition (Vector m, int xleft) {
  Vector xleftrans = new Vector();
  for (int i=0; i<m.size();i++){
    xleftrans.addElement(((Vector)m.elementAt(i)).elementAt(m.size()-xleft));
  }
  return xleftrans;
}

public Vector mlistAfterBreakdown (int t, Vector mlist, int xleft) {
  Vector mlistnew = new Vector();
  Vector temp;
  for (int i = 0; i < mlist.size(); i++) {
    temp = xextractfromMachine(t, ((Vector)mlist.elementAt(i)), xleft );
    mlistnew.addElement(temp);
  }
  return mlistnew;
}

public Vector mfixedAfterBreakdown (int t, Vector mlist) {
  Vector mlistnew = new Vector();
  Vector temp;
  for (int i = 0; i < mlist.size(); i++) {
    temp = extractFixed(t, ((Vector)mlist.elementAt(i)) );
    mlistnew.addElement(temp);
  }
```

```
 return mlistnew;
}

public Vector mlistAfterBreakdown (int t, Vector mlist) {
 Vector mlistnew = new Vector();
 Vector temp;
 for (int i = 0; i < mlist.size(); i++) {
  temp = extractfromMachine(t, ((Vector)mlist.elementAt(i)) );
  mlistnew.addElement(temp);
 }
 return mlistnew;
}

/*public void updateNodesBreakDown (Vector m, int t, int length) {
 for (int i=0; i < m.size(); i++) {
  Node n = ((Node)m.elementAt(i));
  if (n.getRel_time() < t + length) {
   n.setRel_time(t+length);
  }
 }
}*/

 public void updateNodesBreakDown (DirectedGraph g, Vector m, int t, int length) {
 for (int i=0; i < m.size(); i++) {
  Node n = ((Node)m.elementAt(i));
  VertexImpl v = findVertex(g, n);

  if (n.getRel_time() < t + length) {
   Iterator iter = g.getIncomingEdges(v).iterator();
   while (iter.hasNext()) {
    DirectedWeightedEdgeImpl e = (DirectedWeightedEdgeImpl)iter.next();
    double weight = e.getWeight();
    e.setWeight(weight + t + length - n.getRel_time());
   }
  }
 }
}

public int scheduleAffected (int t1, int t2, Vector m) {
 Vector m2 = new Vector();
 for (int i=0; i<m.size();i++) {
  Node n = (Node)m.elementAt(i);
  if (n.getRel_time()<t2 && n.getRel_time()>=t1) {
   m2.addElement(n);
  }
 }
 int t = m2.size();
 return t;
}

public double zeroColumnleft (int t, int mindex, int length, Window win) {

 //Ordonnancement selon la Shift.
 double cmax;
 DirectedGraphImpl g = createGraph(win.plist);
 Vector mlist = win.machines();
 updateData(g, mlist, win.plist);
 Vector mlist0 = shiftB2(g, mlist, (Vector)win.plist);

 int indexbreakdown = machineIndex2(mlist0, mindex);

 int affect = scheduleAffected(t, t+length, (Vector)mlist0.elementAt(indexbreakdown) );

 if (affect == 0) {
  cmax = critpath(g, getsink(g));
  System.err.println("L'ordonnancement pas affecté "+cmax);
  return cmax;
 }

 //Extraction de la partie figée
 Vector mfixed = mfixedAfterBreakdown(t, mlist0);
 mfixed = removeNull(mfixed);

 //Extraction de la nouvelle mlist, selon t
 Vector newmlist = mlistAfterBreakdown(t, mlist0);
 newmlist = removeNull(newmlist);
```

```
  //suppression des arcs dans le graphe avant le re-ordonnancement
  for (int i =0; i < newmlist.size(); i++) {
    Vector m = (Vector)newmlist.elementAt(i);
    delMachine(g, m);
  }

  //Ajustement de la nouvelle mlist
  Vector startleft = startLeft(t, mlist0);
  startleft =removeNull(startleft);
  for (int h =0; h < startleft.size(); h++) {
    Vector m = (Vector)startleft.elementAt(h);
    delMachine(g, m);
  }

  //mise à jour globale
  updateData(g, newmlist, win.plist);

  //mise  à jour en intégrant la panne
  int indexm = machineIndex2(newmlist, mindex);
  updateNodesBreakDown(g, (Vector)newmlist.elementAt(indexm), t, length);

  //mise à jour globale
  updateData(g, newmlist, win.plist);

  //cmax = shiftB(g, newmlist, (Vector)win.plist);
  Vector mfinal = shiftB2(g, newmlist, (Vector)win.plist);

  //Vecteur link

  Vector link = new Vector();
  for (int l = 0; l < mfinal.size(); l++){
    link.addElement(((Vector)mfinal.elementAt(l)).firstElement());
  }

  //Lien entre la fenêtre figée et la fenêtre ré-ordonnancée
  joinWindows(g, mfixed, link);

  //dernière mise à jour
  updateData(g, mfinal, win.plist);

  int merde = 0;

  cmax = critpath(g, getsink(g));
  System.err.println("L'ordonnancement FINAL vaut "+cmax);
  printmlist(mlist0);
  return cmax;
}

public void joinWindows (DirectedGraph g, Vector mlist, Vector link) {
  for (int i = 0; i < link.size();i++) {
    Node n2 =  (Node)link.elementAt(i);
    int index = machineIndex(mlist, (Node)link.elementAt(i));
    if (index!= -1){
      Node n1 = (Node)((Vector)mlist.elementAt(index)).lastElement();
      plusEdge(g, n1, n2);
    }
  }
}

public static void main(String[] args) {
  Scheduler scheduler1 = new Scheduler();
  //Window win = new Window(new File("la06.out"));
  Window win = new Window(new File("D:\\master\\code\\la06.txt"));
  long time  = System.currentTimeMillis();
  scheduler1.ShiftingBottleneck(win);
  System.out.println("Durée Shift. : "+(System.currentTimeMillis()  - time));
  /*time  = System.currentTimeMillis();
  scheduler1.decompositonSB5_2(win);
  System.out.println("Durée Décomp. 2 : "+(System.currentTimeMillis()  - time));
  time  = System.currentTimeMillis();
  scheduler1.decompositonSB5_3(win);
  System.out.println("Durée Décomp. 3: "+(System.currentTimeMillis()  - time));
  time  = System.currentTimeMillis();
  //scheduler1.decompositonSB10_5(win);
  //System.out.println("Durée Décomp. 5: "+(System.currentTimeMillis()  - time));*//}
```

BIBLIOGRAPHY

[1] Adams J, Balas E, Zawack D. The shifting bottleneck procedure for job shop scheduling. Management Science 1988;34:391-401.

[2] Akturk, S., Gorgulu, E., 1998. Match-up scheduling under a machine breakdown. European Journal of Operational Research 112 (1), 80±96.

[3] Applegate D, Cook W. A computational study of the job shop scheduling problem. ORSA Journal of Computing 1991;3:149-56.

[4] Bai, X., Srivatsan, N., and Gershwin, S. B., Hierarchical Real-Time Scheduling of a Semiconductor Fabrication Facility, 1990 IEEE/CMPT International Electronics Manufacturing Technology Symposium, p. 312-317.

[5] Balas E, Lenstra J, Vazacopoulos A. The one-machine problem with delayed precedence constraints and its use in the job shop scheduling. Management Science 1995;41:94-109.

[6] Barnes J, Chambers J. Solving the job shop scheduling problem with tabu search. IIE Transactions 1995;27:257-63.

[7] Bean, J., Birge, J.R., Mittenthal, J., Noon, C.E., 1991. Matchup scheduling with multiple resources, release dates and disruptions. Operations Research 39 (3), 470±483.

[8] Bengu, G., 1994. A simulation-based scheduler for flexible flowlines. International Journal of Production Research 32 (2), 321±344.

[9] Church, L.K., Uzsoy, R., 1992. Analysis of periodic and event driven rescheduling policies in dynamic shops. International Journal of Computer Integrated Manufacturing 5 (3), 153± 163.

[10] Connors, D. P., Feigin, G. E., and Yao, D. D., A Queuing Network Model for Semiconductor Manufacturing, IEEE Transactions on Semiconductor Manufacturing, Vol. 9, No. 3, p. 412-427, August 1996.

[11] Dauzere-Peres S, Lasserre J. A modified shifting bottleneck procedure for job-shop scheduling. International Journal of Production Research 1993;31:923-32.

[12] Dutta, A., 1990. Reacting to scheduling exceptions in FMSenvironments. IIE Transactions 22 (4), 300±314.

[13] El Adl, M. K., Rodriguez, A. A., and Tsakalis K. S., Hierarchical Modelling and Control of Re-entrant Semiconductor Manufacturing Facilities, IEEE Proceedings of the 35th Conference on Decision and Control, Kobe, Japan, December 1996, p. 1736-1742.

[14] Farn, C.K., Muhleman, A.P., 1979. The dynamic aspects of a production scheduling.International Journal of Production Research 17 (15).

[15] Fox, M.S., Smith, S.F., 1984. ISIS ± A knowledge based system for factory scheduling. Expert Systems 1, 25±49.

[16] Girish, M. K., and Riera, J. A., Improvement of Semiconductor Fab Performance with Efficient Scheduling, Japan/USA Symposium on Flexible Automation, Vol. 2, p. 1395-1401, ASME 1996.

[17] He, Y., Smith, M.L., Dudek, R.A., 1994. E.ect of inaccuracy processing time estimation on e.ectiveness of dispatching
rule. In: Third Industrial Engineering Research Conference, pp. 308±313.

[18] Holloway, C.A., Nelson, R.T., 1974. Job shop scheduling with due dates and variable processing times. Management Science 20 (9).

[19] Hsieh, B-W., Chang, S-C., Chen, C-H., Dynamic Scheduling Rule Selection for Semiconductor Wafer Fabrication, IEEE Proceedings of International Conference On Robotics and Automation, Seoul, Korea, May 21-26, 2001.

[20] Hung, Y.-F., and Leachman, R. C., A Production Planning Methodology for Semiconductor Manufacturing Based on Iterative Simulation and Linear Programming Calculations, IEEE Transactions on Semiconductor Manufacturing, Vol. 9, No. 2, p. 257-269, May 1996.

[21] Jain, S., Foley W.J., 1987. Real time control of manufacturing systems with redundancy. Computer and Engineering (2).

[22] Kim, M.H., Kim, Y., 1994. Simulation based real time scheduling in a flexible manufacturing systems. Journal of Manufacturing Systems 13 (2), 85±93.

[23] Kiran, A.S., Alptekin, S., Kaplan A.C, 1991. Tardiness heuristic for scheduling flexible manufacturing systems. Production Planning and Control 2 (3), 228±241

[24] Kubiak, W., Lou, S. X. C., and Wang Y., Mean Flow Time Minimization in Re-entrant Job Shops with a Hub, Operations Research, Vol. 44, No. 5, September October 1996.

[25] Kutanoglu, E., Sabuncuoglu, I., 1994. Experimental investigation of scheduling rules in a dynamic job shop with weighted tardiness costs. In: Third Industrial Engineering Research Conference, pp. 308±312.

[26] Kutanoglu, E., Sabuncuoglu, I., 1998a. Simulation-based scheduling: Part 1: Background and literature review. Technical Report IEOR-9820. Department of Industrial Engineering, Bilkent University, Ankara.

[27] Kutanoglu, E., Sabuncuoglu, I., 1998b. Simulation-based scheduling: Part 2: Experimental study. Technical Report
IEOR-9821. Department of Industrial Engineering, Bilkent University, Ankara.

[28] Lenstra J, Rinnooy Kan A, Brucker P. Complexity of machine scheduling problems. Annals of Discrete Mathematics 1977;1:343-62.

[29] Lou, S. X. C., and Kager P. W., A Robust Production Control Policy for VLSI Wafer Fabrication, IEEE Transactions on Semiconductor Manufacturing, Vol. 2, No. 4, p. 159-164, November 1989.

[30] Lozinski, C., and G1assey, C. R., Bottleneck Starvation Indicators for Shop F100r Contro1,fi IEEE Transactions on Semiconductor Manufacturing, Vol. 1, No. 4, p. 147-153, November 1988.

[31] Lu, S. C. H., Rarnaswamy, D., and Kumar, P. R., Scheduling Semiconductor Manufacturing Plants to Reduce Mean and Variance of Cycle Time, 1993 IEEE/SEMI.

Advanced Semiconductor Manufacturing Conference. p. 83-85.

[32] Matsuura, H., Tsubone, H., Kanezashi, M., 1993. Sequencing, dispatching, and switching in a dynamic manufacturing environment. International Journal of Production Research 31 (7), 1671±1688.

[33] Muhleman, A.P., Lockett, A.G., Farn, C.K., 1982. Job shop scheduling heuristics and frequency of scheduling. International Journal of Production Research 20 (2), 227± 241.

[34] Narahari, Y., and Khan, L. M., fiModeling Re-entrant Manufacturing Systems with Inspection Stations, Journal of Manufacturing Systems, Vol. 15, No. 6, p. 367 378, 1996.

[35] Nof, S.Y., Grant, F.H., 1991. Adaptive/predictive scheduling: review and a general framework. Production Planning and Control 2 (4), 298±312. Society 41 (6), 539±552.

[36] Nowicki E, Smutnicki C. A fast taboo search algorithm for the job shop problem. Management Science 1996;42:797-813.

[37] Ovacik I, Uzsoy R. A shifting bottleneck algorithm for scheduling semiconductor testing operations. Journal of Electronic Manufacturing 1992;2:119-34.

[38] Ramudhin A, Philip Marier. The Generalized Shifting Bottleneck Procedure. European Journal of Operational Research. 93 (1996) 34-48

[39] Singer M. Decompsoition methods for large job shops. Computers and Operations Research 28 (2001) 193-207

[40] Sabuncuoglu, I., Karabuk, S., 1997. Analysis of scheduling± rescheduling problems in a stochastic manufacturing environment. Technical Report IEOR-9704. Department of Industrial Engineering, Bilkent University, Ankara. 27 (9), 1603±1623.

[41] Szelke, E., Kerr, R.M., 1994. Knowledge-based reactive scheduling. Production Planning and Control 5 (2), 124±145.

[42] Taillard E. Parallel taboo search techniques for the job shop scheduling problem. ORSA Journal of Computing 1994;6:108-17.

[43] Uzsoy, R., Perry, C., 1993, Reactive Scheduling of a Semiconductor Testing Facility, IEEE/CHMT Int'l Electronics Manufacturing Technology Sympoium.

[44] Van Laarhoven P, Aarts E, Lenstra J. Job shop scheduling by simulated annealing. Operations Research 1992;40:113-25.

[45] Wein LM, Chevalier PB. A broader view of the job-shop scheduling problem. Management Science 1992;38:1018-33.

[46] Wein, L. M., Scheduling Semiconductor Wafer Fabrication, IEEE Transactions on Semiconductor Manufacturing, Vol. 1, No.3, p. 115-130, August 1988.

[47] Wu, S.D., Wysk, R.A., 1988. Multipass expert control system ± A control/scheduling structure for flexible manufacturing cells. Journal of Manufacturing Systems 7 (2), 107± 120.

[48] Wu, S.D., Wysk, R.A., 1989. An application of discrete-event simulation to on-line

control and scheduling in flexible manufacturing. International Journal of Production Research 27 (9), 1603±1623.

[49] Yamamoto, M., Nof, S.Y., 1985. Scheduling in the manufacturing operating system environment. International Journal of Production Research 23 (4), 705±722.

Vita

Ahmed S. Ghoniem, son of Saad Ghoniem and Hamida Abd El-Aziz, was born on March 1st 1980 in El-Gizah, Egypt. He graduated from Lycée Français de Doha (Qatar) in June 1997. He received an Acadamic Excellence Scolarship by the French government to attend intensive Mathematics and Physics classes preparing for the French engineering school highly competitive exams. In June 1998 he ranked 84 out of more than 7500 students taking the Ecole des Mines de Nantes competitive exams. He enrolled Ecole des Mines de Nantes in 1998. In December 2000, he was selected by Ecole des Mines de Nantes to follow the joint graduate program for a dual Master of Science degree co-developed with the ISE department of Virginia Tech. In July 2002, he graduated from Ecole des Mines de Nantes and received a Master of Science in *Operations Management in Production and Logistics*. This thesis completes his M.S. degree in *Industrial and Systems Engineering* from Virginia Tech.