View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Bilkent University Institutional Repository

ROBUSTNESS and STABILITY MEASURES for SCHEDULING POLICIES in a SINGLE MACHINE ENVIRONMENT

A THESIS
SUBMITTED TO THE DEPARTMENT OF INDUSTRIAL ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCES
OF BILKENT UNIVERSITY
IN PARTIAL FULFULLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Selçuk Gören
September 2002

I certýfy that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

____

Prof. Ýhsan Sabuncuoðlu (Principal Advisor)

I certýfy that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

____

Assoc. Prof. M. Selim Aktürk

I certýfy that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

____

Prof. Erdal Erel

Approved for the Institute of Engineering and Sciences:

____

Prof. Mehmet Baray

Director of Institute of Engineering and Sciences

ABSTRACT

ROBUSTNESS and STABILITY MEASURES for SCHEDULING POLICIES in a SINGLE MACHINE SCHEDULING ENVIRONMENT

Selçuk Gören
M.S. in Industrial Engineering
Supervisor: Prof.Ýhsan Sabuncuoðlu
September 2002

Scheduling is a decision making process that concerns allocation of limited resources (machines, material handling equipment, operators, tools, fixtures, etc.) to competing tasks (operations of jobs) over time with the goal of optimizing one or more objectives. The output of this decision process is time/machine/operation assignments. In classical scheduling theory, the objective is generally maximizing some measure of system performance. In addition to classical performance measures two new criteria are used in modern scheduling literature: "robustness" and "stability". In this thesis, we propose several robustness and stability measures and policies. Two new surrogate measures are also developed since the exact measures are difficult to calculate. These surrogate measures are embedded in a tabu search algorithm to generate robust and stable schedules for a single machine subject to random machine breakdowns. We show that our surrogate measures are better than well-known and commonly used average slack method.

Keywords: Robustness, Stability, Single Machine Scheduling

ÖZET

TEK MAKINALI ORTAMDA ÇÝZELGELEME POLÝTÝKALARIÝÇÝN SAÐLAMLIK ve KARARLILIK ÖLÇÜTLERÝ

Selçuk Gören
Endüstri Mühendisliði Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Ýhsan Sabuncuoðlu
Eylül 2002

Çizelgeleme sýnýrlý kaynaklarýn (makinalar, malzeme taþýnmasý donanýmý, operatörler, kesici uçlar, vb.) ayný kaynaklara ihtiyacý olan deðiþik iþlere tahsis edilmesini ilgilendiren bir karar verme sürecidir. Çizelgeleme, bir veya daha fazla amaç fonksiyonunu eniyilemek için yapýlýr. Bu karar verme sürecinin sonucu olarak zaman / makina / iþ atamalarý elde edilir. Klasik çizelgeleme teorisinde amaç fonksiyonu genellikle herhangi bir sistem performans ölçütünü eniyilemektir. Modern çizelgeleme literatüründe ise klasik sistem performans ölçütlerine ek olarak iki yeni kriter daha ele alýnmaktadýr: "saðlamlýk" ve "kararlýlýk". Bu tezde saðlamlýk ve kararlýlýðý tanýmlayarak çeþitli ölçütler ve saðlam ve kararlý çizelgeleme politikalarý geli°tirmekteyiz. Çoðu durumlarda bu ölçütlerin tam olarak hesaplanmasý mümkün olmadýðý için bunlarýn yerine geçebilecek iki ölçüt geliþtirildi. Bu ölçütler bir tabu arama algoritmasý ile birlikte kullanýlarak rassal arýzalanmalara tabi olan tek makinalý çizelgeleme problemi için saðlam ve kararlý çizelgeler oluþturduk. Yeni ölçütlerimizin, literatürde ayný amaç için sýkça kullanýlan "ortalama gevþeklik" yönteminden daha iyi sonuçlar verdiðini gördük.

Anahtar Sözcükler: Saðlamlýk, Kararlýlýk, Tek Makinalý Çizelgeleme

To my family…

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Prof. Ýhsan Sabuncuoðlu for his invaluable guidance, remarks and recommendations during my graduate study. He has been supervising me with patience and his great helps bring this thesis to an end.

I am also indebted to Assoc. Prof. M. Selim Aktürk and Prof. Erdal Erel for accepting to read and review this thesis and for their suggestions.

I would like to thank to Hakan Gültekin, Pýnar Zarif Tan and Hakan Ümit for their friendships. Without their academic and most importantly morale support I would not be able to bear all.

I also would like to thank to Hüseyin Bilal Pandül for his friendship since our undergradute years. I am so grateful for his support.

Finally, I would like to express my deepest gratitude to my family for their continuous morale support.

Contents

1. Introduction ... 1

2. Problem Definition and Research Methodology ... 4

2.1. Definition of Reactive Scheduling Problem, Notations and Terms ... 4

2.2. Robustness and Stability Policies and Measures ... 6

2.2.1. Robustness Policies and Measures ... 6

2.2.1.1. Robustness Policies and Measures Based on Realized Performances ... 6

2.2.1.2. Robustness Policies and Measures Based on Differences between Actual and Optimal Performances ... 7

2.2.2. Stability Policies and Measures ... 8

2.3. Frequencey and Method of Scheduling ... 11

2.4. Disruptions and Possible Responses ... 16

3. Literature Review ... 20

4. Observations and Proposed Methods ... 35

4.1. Observations ... 35

4.2. Proposed Methods ... 36

4.2.1. The Proposed Algorithm ... 37

4.2.2. Tabu Search ... 39

4.2.3. Neighbourhood Evaluation Methods ... 41

4.2.3.1. Method 1 ... 41

4.2.3.2. Method 2 ... 43

4.2.4. A Numerical Example 44
5. Experimental Study 49
5.1. Experimental Environment 49
5.1.1. Problem Parameters 49
5.1.2. Breakdown Parameters 51
5.2. Fine Tuning Algorithm Parameters 52
5.3. Comparison of Method 1 with Other Approaches 56
5.3.1. Robustness 56
5.3.2. Stability 59
5.3.3. Bicriterion Approach 62
5.3.4. Robustness vs Stability 65
5.4. Comparison of Method 2 with Other Approaches 67

6. Conclusion 72

Bibliography 75

Appendix 79

List of Figures

1. Scenario Based Representation of Disruptions 5
2. The Possibilities for Number of Scenarios.. 10
3. Full/partial Scheduling 13
4. Illustration of Proposed Algorithm.. 39
5. Parameters of Method 1 42
6. Estimation of Realization under Method 1 43
7. Parameters of Method 2 44
8. L and U Parameters of Method 1 46
9. Illustration for Numerical Example.. 47
10. Estimate vs Realized Performance Measure plots (Robustness) 57
11. Estimate vs Realized Performance Measure plots (Stability)....................................... 61
12. Estimate vs Realized Performance Measure plots (Bicriterion) 63
13. Robustness vs Stability.. 66
14. Estimate vs Realized Performance Measure Plots (Continuous Scheduling)............... 69

List of Tables

1. Scheduling Policies 14
2. Disruptions and Possible Responses 18
3. Cross Table for Type of Response/When to Schedule 19
4. Classification of Studies 33
5. Job Parameters for the Numerical Example 44
6. Evaluation of Neighbourhood 48
7. Calculation of Average System Slack 48
8. Calculation of Robustness Measure of Method 2 48
9. Test Problem Parameters 51
10. Breakdown Parameters 52
11. The Maximum Number of Iterations between Best Solution Improvements 53
12. Tabu Tenure Comparisons 54
13. Summary table for robustness results 58
14. Summary table for stability results 60
15. Summary table for bicriterion approach results 64
16. Composite Objective Function (0.85*Robustness + 0.15*Stability) Values 65
17. Realized Performance Measures under Method 1 (Summary) 65
18. Stability Measures under Method 1 (Summary) 66
19. Summary Table for Continuous Scheduling Results (Robustness) 68
20. Comparison of λ Values - Total Tardiness Case 80
21. Comparison of λ Values - Total Tardiness Case (Error Percentage - R^2 values) 82
22. Comparison of λ Values - Total Flow Time Case 84
23. Comparison of λ Values - Total Flow Time Case (Error Percentage - R^2 values) 86
24. Makespan Results (Robustness) 88
25. Total Tardiness Results (Robustness) 89
26. Total Flow Time Results (Robustness) 90

27. Makespan Results (Stability) 91
28. Total Tardiness Results (Stability) 92
29. Total Flow Time Results (Stability) 93
30. Makespan Results (Bicriterion) 94
31. Total Tardiness Results (Bicriterion) 95
32. Total Flow Time Results (Bicriterion) 96
33. Makespan Results (Continuous Scheduling) 97
34. Total Tardiness Results (Continuous Scheduling) 98
35. Total Flow Time Results (Continuous Scheduling) 99

Chapter 1

Introduction

Scheduling is a decision making process that concerns allocation of limited resources (machines, material handling equipment, operators, tools, fixtures, etc.) to competing tasks (operations of jobs) over time with the goal of optimizing one or more objectives (Pinedo, 1995). The output of this decision process is time/machine/operation assignments.

In classical scheduling theory, the objective is generally optimizing system performance. The performance of a given schedule is generally assessed by some measures based on completion times or due-dates of the jobs. Among these are maximum flow time, maximum completion time (or makespan), mean flow time, mean completion time, maximum lateness, total tardiness, mean lateness, mean tardiness, etc. These performance measures are all *regular.* A *regular performance measure* is non-decreasing in completion times; that is, if any job is made to finish later, the measure stays the same or increases but never decreases. In classical approach, a schedule is generated with the objective of optimizing one or more of these performance measures.

Recently, Kempf, Uzsoy, Smith, and Gary (2000) propose a different approach: *segmentation* and then *aggregation* of the metrics that are used to measure the performance of a schedule. *Segmentation* is specifying classes of scheduling objects that form a meaningful unit (e.g. all drilling machines in a plant can form a segment). Once the segmentation of the scheduling objects has been specified (i.e. the scheduling systems is divided into several segments), the metrics for each segment (*e.g.*. utilization of machines in that segment) can be *aggregated* into one segment-wide metric. The

authors propose to use different metrics for each segment, rather than using a single metric for the whole schedule. For example, a schedule that maximizes utilization of the machines throughout the whole system may be of poor quality, because this schedule would increase work-in-progress inventory levels. Instead, a good schedule should maximize utilization of only the bottleneck machines (where high utilization is needed).

In practice due to unexpected disruptions (breakdowns, order cancellations, etc.), schedules either become infeasible or invalid so quickly that they need to be modified in an appropriate manner. In fact, the ignorance of these inevitable disruptions in scheduling process can be viewed as the major source of the gap between scheduling theory and practice. During the last decade, several authors investigate possible ways to cope with disruptions in the scheduling process. In general, this approach is called as *reactive scheduling*. In the reactive scheduling literature, in addition to classical performance measures there are two new criteria used in scheduling decisions, "robustness" and "stability". A schedule whose performance does not degrade much in the face of disruptions is called *robust*. The performance of a robust schedule should be insensitive to disruptions. When evaluating schedules, performance of the realized schedule is more important than planned or estimated performance of initial schedule, because the former is the reality while the latter is just an anticipated course of actions. A schedule whose realized schedule does not deviate from the original schedule in the face of disruptions is called *stable*. Beside allocating machines to competing jobs, a schedule also serves as a plan for other activities, such as determining shipments dates, releasing orders to suppliers, planning requirements for secondary resources such as tools, fixtures, etc. Deviations from the planned schedule disrupt these secondary plans and create system nervousness. Thus, adhering to the original schedule during the execution (i.e. stability of the schedule) is desirable.

In summary, random disruptions make two types of hazards to the schedules during the execution: degrading schedule performance, and increasing system nervousness by causing deviations from the planned schedule. In the literature, there are some researchers who have attempted to generate robust and stable initial schedules

when the response method to disruptions is known (Leon, Wu and Storer (1994), Mehta and Uzsoy (1998), etc.). On the other hand, some authors have developed good response methods to disruptions (Wu, Storer, and Chang (1993), Wu, Byeon, and Storer (1999), etc.).

In this study, we propose several robustness and stability measures and policies. Since the exact measures are difficult to calculate, we also develop two new surrogate measures as an alternative to well-known and commonly used average slack method. These new surrogate measures are embedded into a Tabu Search algorithm to generate robust and stable schedules for a single machine subject to stochastic breakdowns. Our extensive computational experiments indicate that the new surrogate measures are better than the average slack method.

The remainder of this study is organized as follows: In the next chapter, we provide some definitions and notations. Chapter 3 is devoted to an extensive review of the literature. In Chapter 4 we consider the single-machine scheduling environment subject to random machine breakdowns, and provide a solution methodology that involves two new surrogate measures for this problem. In Chapter 5 we conduct extensive computational experiments to assess the performance of our methodology, whereas the last chapter is devoted to the concluding remarks and future research directions.

Chapter 2

Problem Definition and Research Methodology

2.1 Definition of Reactive Scheduling Problem, Notations and Terms

Let P_0 be our original scheduling problem. Suppose we have m feasible solutions or schedule alternatives, S_1 through S_m. In most of the scheduling problems, m is quite large (infinitely many). In classical scheduling theory, our aim is to find the optimal schedule, S_0^*, according to a performance measure such as maximum lateness, maximum completion time, average flow time, etc. Let $f(S)$ denote the value of this performance measure for schedule S. Then, we select the schedule S_0^* such that $f(S_0^*) \leq f(S_i)$ for $i = 1, 2 \ldots m$, without loss of generality. In what follows, we consider possible disruptions during the schedule generation process.

Suppose that there are n scenarios corresponding to possible disruptions in the course of execution. The original problem P_0 changes into P_j for scenario j with probability a_j. Accordingly, schedule S_i changes into schedule S_{ij} under j^{th} scenario. That is, S_{ij} is the realized schedule of the initial schedule S_i, if the disruptions in scenario j occur. Let S_j^* be the optimal solution to the problem P_j, with the objective function value of $f(S_j^*)$. Figure 1 illustrates all these ideas.

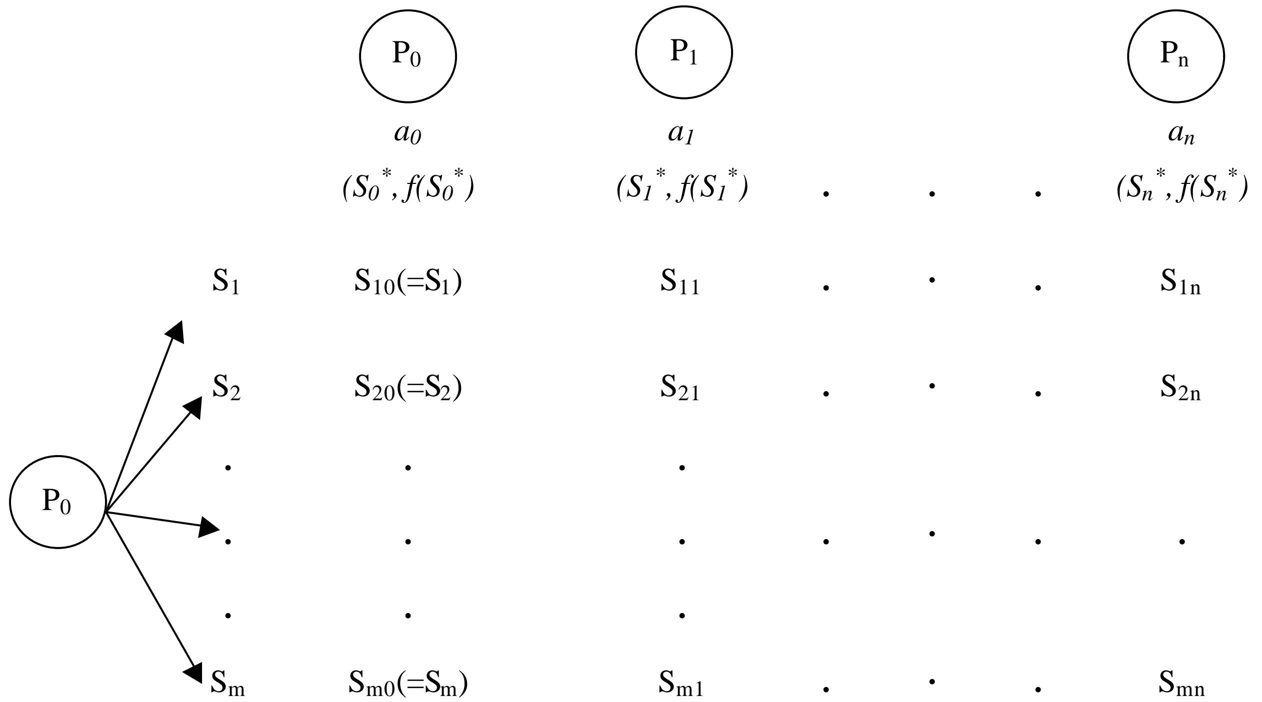


Figure 1. Scenario Based Representation of Disruptions

If we knew that scenario j would occur, we would try to find schedule S_j^*. However, we do not know which scenario will actually occur. The problem is further complicated by the fact that it may be extremely difficult to find S_j^* for P_j due to the complexity of scheduling problems. As explained before, there are numerous performance measures used in the scheduling literature (tardiness, flow time, makespan, etc.). In the next section, we introduce possible robustness and stability measures.

2.2. Robustness and Stability Policies and Measures

2.2.1. Robustness Policies and Measures

Some robustness measures are based on the actual performances of the realized schedules, $f(S_{ij})$, and some are based on the differences between actual and optimal performances, $f(S_{ij}) - f(S_j^*)$. The former aims at selecting a schedule with a good realized performance, whereas the latter tries to select a schedule whose performance is not bad relative to the best performance.

2.2.1.1. Robustness Policies and Measures Based on Realized Performances

1. Select S_i that minimizes the expected realized performance, $E[f(S_i^r)]$, where $E[f(S_i^r)] = \sum_{j=0}^{n} a_j f(S_{ij}),\ for\ i = 1,2,\ldots,m.$ This method selects the schedule whose performance measure is the best on the average. This is a risk neutral approach. (Wu *et al.*, 1999)
2. Select S_i that minimizes the worst-case scenario performance. That is select S_i such that $\max_j \{f(S_{ij})\}$ is minimized over all i (i.e $\min_i \{\max_j \{f(S_{ij})\}\}$). This policy selects the schedule whose worst-case performance measure is better than all others' are. This is a risk-averse approach.
3. If the decision-maker can identify the worst case scenario, say P_k, k **Î** *{1,2...n}*, selecting the schedule that performs the best in P_k can be an appropriate policy. Thus, this policy selects S_i with $\min_i \{f(S_{ik})\}$ where P_k is the worst case scenario.
4. Similarly, if the decision-maker can identify the most probable scenario, that is the scenario P_m where $m = \arg\max_i \{a_i\}$, for $i = 1, 2, \ldots n$, then selecting the schedule whose performance measure is the best under P_m can be an appropriate

policy. Thus, this policy selects S_i with $\min_i \{f(S_{im})\}$ where P_m is the most probable scenario.

5. Another policy is to select the schedule such that expected deviation of realized schedule's performance from the initial anticipated performance is minimized. That is, select S_i with minimum $\boldsymbol{s}_i = | f(S_{i0}) - E[f(S_i^r)] |$, for $i = 1,2, \ldots m$. This policy emphasizes the definiton "robust schedule is the schedule whose performance does not degrade much in the face of disruptions".
6. Another policy is to select the schedule that minimizes a measure that is a convex combination of aforementioned measures. For example, selecting the schedule which minimizes $r.E[f(S_i^r)] + (1-r).\boldsymbol{s}_i$, where r is a real number between 0 and 1. The first part of above measure emphasizes that a robust schedule should perform well in the face of disruptions. The second part emphasizes that a robust schedule's performance should not degrade much in the face of disruptions. By varying r between 0 and 1, different weights can be given to these two points of view of robustness. (Leon *et al.*, 1994)

2.2.1.2. Robustness Policies and Measures Based on Differences between Actual and Optimal Performances

First, construct the differences matrix, $\boldsymbol{D} = [\boldsymbol{d}_{ij}]_{mxn}$, where $\boldsymbol{d}_{ij} = | f(S_{ij}) - f(S_j^*) |$, for $i = 1, 2 \ldots n$ and $j = 1, 2 \ldots n$. The ij^{th} element of this matrix is the difference between optimal performance of jth scenario and the realized performance of ith schedule in this scenario.

1. Select S_i that minimizes expectation of differences, $E[\boldsymbol{d}_i]$, where $E[\boldsymbol{d}_i] = \sum_{j=1}^{n} a_j \boldsymbol{d}_{ij}$, for $i = 1,2, \ldots m$. This method selects a schedule whose performance measure degradation is the least on the average. This is a risk neutral approach.

2. Select S_i that minimizes the worst case scenario performance degradation, that is select S_i such that $\max_j\{\delta_{ij}\}$ is minimized over all i (i.e. $\min_i\{\max_j\{\delta_{ij}\}\}$). This policy selects the schedule whose worst-case performance measure degradation is less than all others. This is a risk-averse approach (Daniels and Kouvelis, 1995).
3. If the decision maker may identify the worst case scenario, say P_k, $k \in \{1,2...n\}$, selecting the schedule whose performance degrades the least in P_k can be an appropriate policy. Thus, this policy selects S_i with $\min_i\{\delta_{ik}\}$, where P_k is the worst case scenario.
4. If the decision-maker can identify the most probable scenario, that is the scenario P_m where $m = \arg\max_i\{a_i\}$, for $i = 1, 2, \ldots n$, then selecting the scenario whose performance degradation is the least under P_m can be an appropriate policy Thus, this policy selects S_i with $\min_i\{\delta_{im}\}$, where P_m is the most probable scenario.

2.2.2. Stability Policies and Measures

Determining the impacts of schedule change is a difficult task. As often suggested in the literature, we use completion time differences to account the impact of schedule change (i.e stability). Let C_k^S be the completion time of job k under schedule S. Let N be the number of jobs. Construct the completion time differences matrix $D = [d_{ij}]$ where $d_{ij} = \sum_{k=1}^{N} | C_k^{S_{ij}} - C_k^{S_{i0}} |$.

1. Select schedule S_i that minimizes the expectation of differences, $E[d_i]$, where $E[d_i] = \sum_{j=1}^{n} a_j d_{ij}$. This method selects a schedule whose stability is the best on the average. This is a risk neutral approach. (Mehta and Uzsoy (1998), Mehta and Uzsoy (1999), O'Donovan, Uzsoy, and McKay (1999)).

2. Select S_i that minimizes the worst case scenario completion time differences, that is select S_i such that $\max_j \{d_{ij}\}$ is minimized over all i. This policy selects a schedule whose worst case stability is greater than all others. This is a risk-averse approach.
3. If the decision-maker can identify the worst-case scenario, say P_k, $k \in \{1,2, \ldots n\}$, selecting the scenario whose stability is the best in P_k. can be an appropriate policy. Thus, this policy selects S_i with $\min_i \{d_{ik}\}$, where P_k is the worst case scenario.
4. If the decision maker may clearly identify the most probable scenario, that is the scenario P_m where $m = \arg\max_i \{a_i\}$, for $i = 1, 2, \ldots n$, then selecting the scenario whose stability is the best under P_m can be an appropriate policy. Thus, this policy selects S_i with $\min_i \{d_{im}\}$, where P_m is the most probable scenario.

If the number of scenarios is finite, then calculation of robustness and stability measures, and thus application of the above policies is easy. However, in practice, the scenarios cannot be easily defined, or known in advance. In addition, the number of scenarios can be infinitely many. Moreover, the number of alternative schedules are too many in practice. Thus, computational burden of calculating robustness or predictability measures can be quite high. In this case one reasonable alternative is to find a good surrogate measure and an efficient scheduling algorithm should be constructed so that the selected schedule optimizes this surrogate measure. Sometimes, although the number of the scenarios is infinitely many, a procedure to generate the best/worst case scenario can be available. In such cases, there is no need to define a surrogate measure (See Daniels and Kouvelis 1995, for such an example). Figure 2 illustrates these possibilities of number of scenarios.

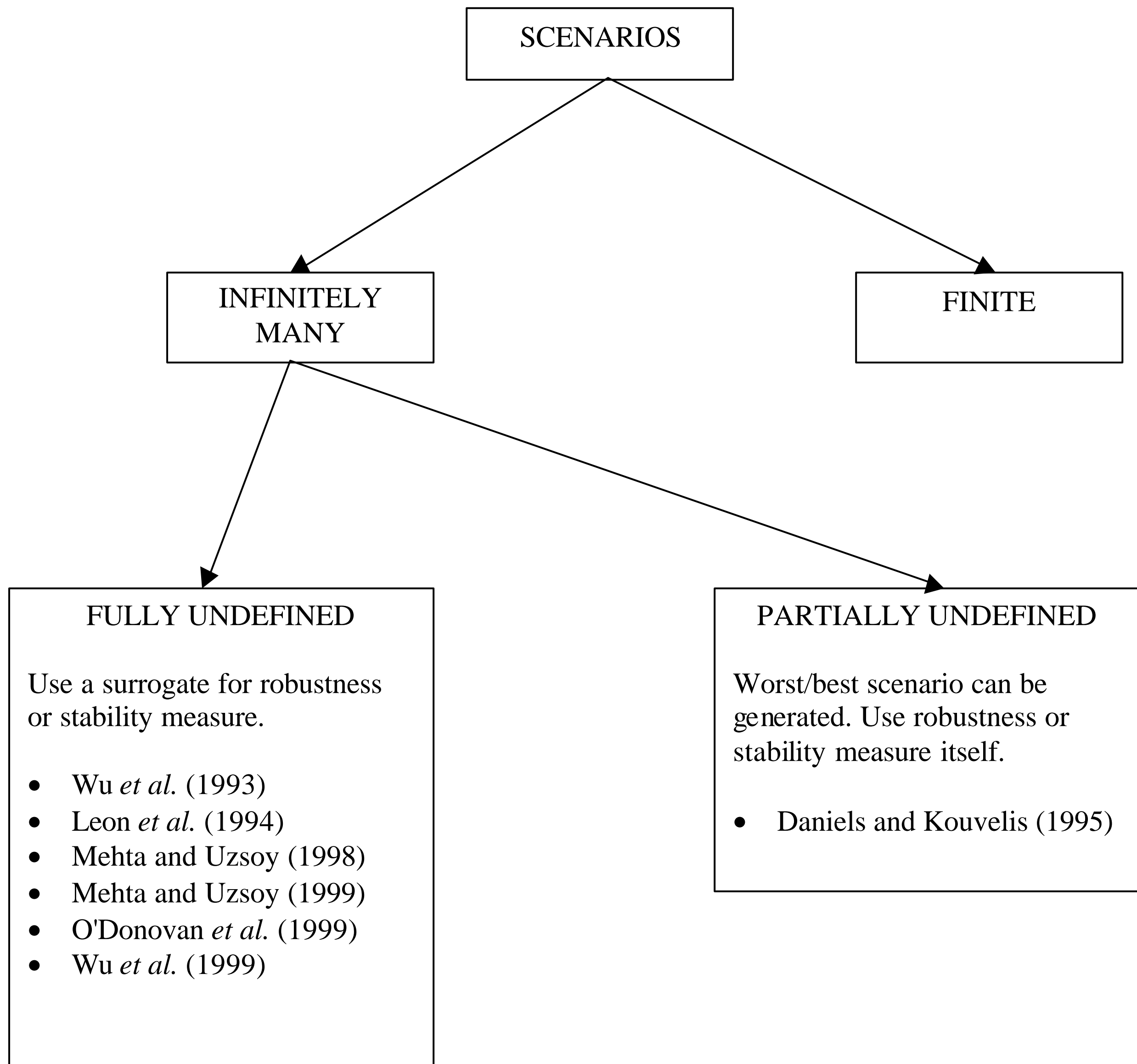


Figure 2. The Possibilities for Number of Scenarios

2.3. Rescheduling Frequency and Methods of Scheduling

Schedules generated in practice cannot be used for a long time period because of unexpected disruptions and random events. Thus, it is necessary to revise the existing schedule at some points in time. Two immediate questions arise: when-to-schedule, and how-to-schedule?

When-to-schedule decision determines the system responsiveness to various kinds of disruptions. As scheduling frequency increases, the system responsiveness also increases. There are several alternative ways to decide on timing of schedule decisions. The first alternative is to schedule the system periodically; this is called as *periodic scheduling*. Here, the period length can be constant or variable. In the constant case, which is quite often used in practice, revisions are made at constant or fixed time intervals. However, according to the variable time interval method, scheduling decisions are made after a certain amount of schedule is realized (See Sabuncuoglu and Karabuk, 1999). Another alternative could be to revise the schedule after a number of events that change system state occur. For example, the schedule can be updated after each machine breakdown, or a new job arrival, etc. This method is usually called as *continuous scheduling*. Another method is *adaptive scheduling*, which may also be called as *controlled response*. According to this policy, a scheduling decision is triggered after a predetermined amount of deviation from the original schedule is observed. For example, revisions can be made when completion time differences between the initial and realized schedules exceeds a threshold value, say 30 minutes, or a predetermined percentage of the assumed makespan. Similarly, the schedules can be revised after a certain amount of deviation from the planned throughput, or flow time. In addition, several hybrid methods, which are combinations of above, can be considered. For example, Yamamoto and Nof (1985) propose a scheduling policy that is called as the *event-driven scheduling*, where revisions are made at the end of fixed time intervals (periodic scheduling), but scheduling is also triggered in response to important events that changes the system state (continuous scheduling).

How-to-schedule decision determines the ways that the schedules are revised or updated. There are also three related issues: The first issue has to do with the scheduling scheme used. *Off-line scheduling* refers to scheduling all operations of available jobs for the entire scheduling period, before execution of schedule, whereas *on-line scheduling* is to take scheduling decisions one at a time, during execution of schedule (See Sabuncuoglu and Hommertzheim, 1992). A good example for on-line scheduling is using well-known priority dispatching rules. Between these two extremes, another alternative could be *quasi-online scheduling*, in which a subset of the operations of the job set are scheduled and the rest is left for future time periods.

The second issue is the amount of data used during the schedule generation process. Kutanoglu and Sabuncuoglu (2001), define forecasting horizon (*FH*) as the time span of job release data. It represents the maximum time period for which we have enough information to generate the schedule. They define look-ahead window (*LW*) as the portion of *FH* for which a new schedule is generated or a revision is made. It can be smaller than *FH* or equal to *FH*. If it is smaller than *FH*, only a part of the available information is used. This is generally due to low confidence about the accuracy of the far-future information. In this case only near-future information is used. If *LW* is equal to *FH*, all available information is used. The approach where *LW=FH* is called as *full* scheduling. The other approach where *LW<FH* is called as *partial* scheduling. Full and partial scheduling is illustrated in Figure 3.

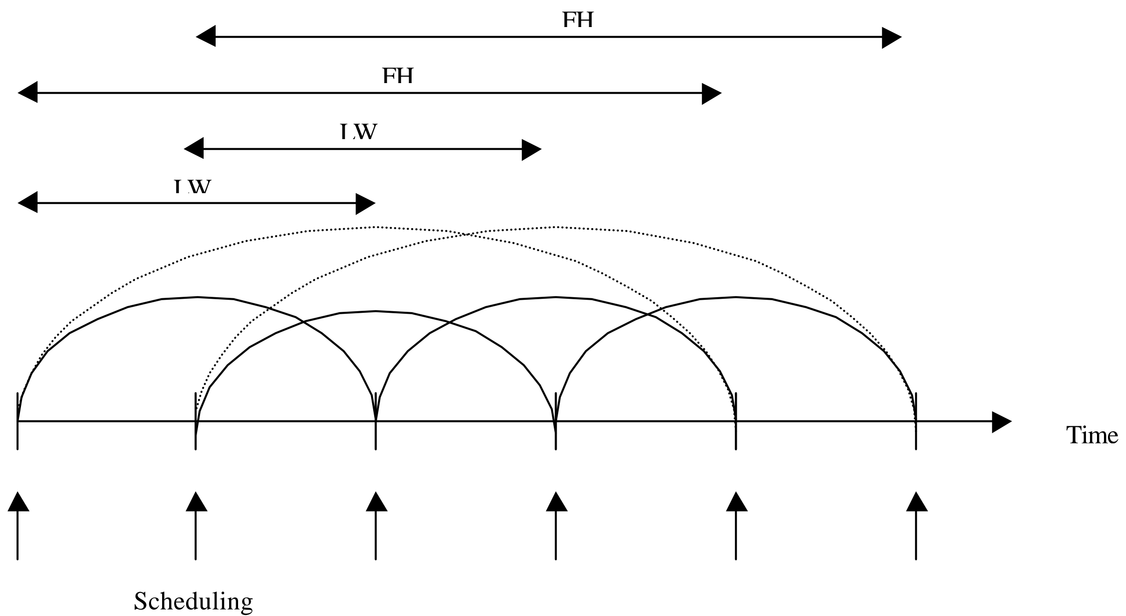


Figure 3. Full/Partial Scheduling

Note that if a full scheduling is employed with periodic review, and if *FH* is equal to period length, this policy corresponds to doing nothing as type of the response (i.e., leaving the system alone and letting it the recover from the disruptions by itself).

Another issue to be discussed is the type of response. One can identify the following two extreme cases: 1) reschedule the operations of all the remaining jobs from scratch, and 2) take no corrective action and let the system recover itself from the negative effects of disruptions. Between these two extremes, one can prefer to repair schedules. One method of repairing could be to generate a match-up schedule, where at some point in future, the new schedule and the original one are the same (i.e. they match with each other). Another method is right/left shifting the remaining jobs altogether in the time horizon so that the disruption length is accommodated. This method does not change the sequence of the jobs. Another minor revision could be to change only a small number of jobs' positions in the schedule. All these when-to-schedule and how-to-schedule alternatives are summarized in Table 1.

Table 1. Scheduling Policies

When to Schedule (Type and Level of Responsiveness)

- Periodic Scheduling (Response)
  - Fixed Time Interval
  - Variable Time Interval (based on amount of the schedule realized)

- Continuous Scheduling (Response)
  - Breakdowns (BR)
  - Rush Orders (RO)
  - Job Arrivals (ARR)
  - Job Completions (JC)

- Adaptive Scheduling (Controlled Response)
  - ΔS (Deviation from the previous schedule)
    - absolute completion times deviations (percentage)
    - throughput or flow time deviations

- Combinations of Above
  - Event Driven Scheduling (Yamamoto and Nof, 1985)
    - Periodic (Fixed Time Interval) + Continuous

Table 1. (Cont'd)

How To Schedule(Type and Method of Response)

• Scheduling Scheme
    • off-line
    • quasi-online
    • On-line(dispatching)
 	
• Amount of Data
    • Full
    • Partial
 	
• Type of Response
    • Do nothing
    • Rescheduling
    • Repair
        • Match-up
        • Right-shift/Left-shift
        • Other minor revisions (insert, append, etc.)
• Performance Metrics
    • Planned (initial) performance measure
    • Stability
    • Robustness

2.4. Disruptions and Possible Responses

In practice, there may be various random or unexpected events that can affect the performance of system. These disruptions are: unexpected job arrivals, machine breakdowns, processing time variability, due-date changes, order cancellations, ready time changes, rush orders and scrap jobs.

In Table 2, we give a list of disruptions and possible responses that can be used to cope with these unexpected events. The first column is the list of disruptions. We classify responses in three categories: *Do nothing* - taking no corrective action, *Reschedule* - rescheduling all operations of available jobs from scratch, and *Repair* - making minor modifications on the existing schedule. In this table, we also present response types for each disruption.

Table 3 cross checks these policies with when-to-schedule decisions. The "X" entry in the cell means that this combination of when-to-schedule and response type can be used together, otherwise the cell is left empty. For example, according to periodic scheduling, revisions are made only after a fixed or variable time period pass after the previous scheduling point. So, if some disruption occurs in between, we take no corrective action and wait for the next scheduling point. Therefore, only ARR1 (Do nothing) response is applicable if the when-to-schedule decision is periodic scheduling. The corresponding cell has an "X" entry and other cells corresponding to responses to unexpected arrivals (ARR2, ARR3 and ARR4) are left empty. As an another example RO2 response inserts the rush order to the first available position in the existing schedule. Again, if our when-to-schedule policy is periodic scheduling, we take no corrective action and wait for the next scheduling point. Thus, RO2 cannot be used with periodic scheduling, so the corresponding cell is left empty. However, if when-to-schedule policy is continuous/RO scheduling (which responds to every rush order) or adaptive scheduling (which reacts after a certain amount of deviation from the planned

schedule is realized) RO2 can be used to react to rush order(s). Hence, the corresponding cells are marked with an “X” character.

Table 2. Disruptions and Possible Responses

Disruptions	Do nothing	Repair	Reschedule
Unexpected Arrivals	Wait until the next scheduling point (ARR1)	• Append at the end of existing schedule (ARR2) • Insert in a suitable position in the existing schedule (ARR3)	Reschedule all the available jobs from scratch (ARR4)
Machine Breakdowns	Right Shift (BR1)	• match-up schedule (BR2)	Reschedule all the available jobs from scratch (BR3)
Processing time variability	Right Shift/Left-Shift (PV1)		Reschedule all the available jobs from scratch (PV2)
Due-Date changes	Wait until the next scheduling point (DD1)	• Find a suitable position for the job whose due-date has been changes (DD2)	Reschedule all the available jobs from scratch (DD3)
Order cancellations	Make to stock (OC1)	• Delete the job that is cancelled in the existing schedule, left shift the portion after the deleted job (OC2)	Reschedule all the available jobs from scratch(OC3)
Ready-time changes	Left or Right Shift (RC1)	• Find a suitable position whose ready time has been changed in the existing schedule (RC2)	Reschedule all the available jobs from scratch (RC3)
Rush orders		• Add the rush order to the beginning of the existing schedule (RO1) • Insert the rush order to the first available position in the existing schedule (RO2)	Reschedule all the available jobs from scratch (RO3)
Scraps and waste	Wait until the next scheduling point, consider a new job in place of the scrap/waste then (SW1)	• Append a new job in place of the scrap/waste at the end of the existing schedule (SW2) • Insert a new job in place of the scrap/waste into the existing schedule (SW3)	Reschedule all the available jobs from scratch (SW4)

Table 3. Cross Table for Type of Response/When to Schedule

Type of Response How To	When To					
	Periodic	Adaptive	Continuous			
			BR	RO	ARR	JC
ARR1	X	X	X	X	X	X
ARR2					X	
ARR3					X	
ARR4					X	
BR1	X	X	X	X	X	X
BR2		X	X			
BR3		X	X			
PV1	X	X	X	X	X	X
PV2		X				
DD1	X	X	X	X	X	X
DD2		X				
DD3		X				
OC1	X	X	X	X	X	X
OC2		X				
OC3		X				
RC1	X	X	X	X	X	X
RC2		X				
RC3		X				
RO1	X	X	X	X	X	X
RO2		X		X		
RO3		X		X		
SW1	X	X	X	X	X	X
SW2		X				
SW3		X				
SW4		X				

Chapter 3

Literature Review

Aytug, Lawley, McKay, Mohan, and Uzsoy (2001) provides a literature review of last decade on scheduling in the face of uncertainties. The authors begin by stating different purposes scheduling. They continue by giving a four-dimensional taxonomy of the uncertainty faced in scheduling environments: *cause, context, impact,* and *inclusion* of disruptions. *Cause* can be viewed as *object* (e.g. machine) and *state* (e.g. not ready). *Context* refers to the environmental situation, that is the factors that can alter expectations for processing time, yield, or any other performance measure. *Impact* refers to the result of the disruption. *Inclusion* refers to the type of handling the disruptions; if the disruptions are considered while generating the schedule, the *inclusion* is *predictive.* On the other hand, if the disruptions are responded after they occur, the *inclusion* is *reactive.* On reviewing the existing studies, the authors make some observations: in most of the existing literature, the *cause* is often machine availability, and the *context* is ignored. The *inclusion* is generally predictive/reactive. The reconfiguration cost of the system after a disruption is not included. The authors show inclusion of *impact* and *context* of uncertainty as well as estimation of reconfiguration costs as future research directions. They also declare using available information on the nature of disruptions, such as using distributions of machine failures, as a fruitful topic for further research.

In what follows, we make our own brief review of the the relevant studies in the literature. Table 4 is prepared to summarize these studies using our classification framework.

Wu *et al.* 's (1993) objective is to reschedule the system (a single machine) after a single disruption (machine failure) such that makespan of the new schedule and stability (e.g. deviation of new schedule from the original schedule in terms of job starting times) are optimized. The authors use pairwise swapping methods and a genetic algorithm to obtain non-dominated solution sets (i.e. a set of non-dominated schedules). Their computational experiments indicated that it is quite possible to improve the stability substantially with slight increases in makespan. The details of this study are given below:

Wu *et al.* (1993) study single machine rescheduling problem under machine disruptions. They reschedule the jobs in response to each machine failure so that minimum makespan is achieved with high schedule stability. Note that these two goals often conflict; minimizing makespan usually requires some changes in the schedule, but high schedule stability can be achieved by minimizing the number of changes. Unlike a typical system performance such as makespan, measuring schedule stability is a difficult task. They consider two criteria for stability. The first one is the deviation of revised schedule in terms of job starting times. This criterion is useful if secondary resources such as tooling, fixtures are delivered according to the original schedule. The second one is the deviation of revised schedule from the original schedule in terms of sequence of the jobs. This criterion is useful if there are sequence dependent tooling, fixtures, set-ups, etc. The measure for the first criterion is the average absolute deviation from the original job starting times, which can be easily calculated. However, the second criterion is difficult to measure. One possible surrogate measure for second criteria can be the sum of absolute deviations of job starting times from the *right-shift* schedule. Recall that right-shift schedule maintains the original sequence and represent minimal disruption to the original schedule when job sequence is important. Therefore, they have two problem types that differ from one another with respect to the stability criterion in use. Since the problem is NP-hard even without stability considerations, they use some heuristic algorithms to solve it (i.e. generate a new schedule in response to a machine breakdown). First type of heuristics is based on the pairwise swapping methods. The second type is local search that utilizes genetic algorithms. All heuristics

results in a list of non-dominated schedules. In pairwise swapping methods, they start with an efficient schedule (that minimizes makespan) without taking stability into account and a stable schedule (that minimizes cost impact of schedule changes) without taking makespan into account. The aim is to find the schedules between these two extremes, thus finding an efficiency frontier.

In the first one of pairwise swapping algorithms (called *r-grid search*), the objective function is defined as a convex combination of makespan and the relevant deviation measure. That is, the objective function is $Z = (r).M(\boldsymbol{p}) + (1-r).D(\boldsymbol{p})$, where $\boldsymbol{p}$ is a schedule, r is a real number between 0 and 1, $M(\boldsymbol{p})$ is the makespan of schedule $\boldsymbol{p}$, and $D(\boldsymbol{p})$ is the deviation measure of schedule $\boldsymbol{p}$. They begin with the efficient schedule by using Carlier's algorithm (Carlier, 1982) and $r = 0.05$. At each step, they generate neighbors from current sequence at hand by swapping places of two jobs. They find the best neighbor and add it to global non-dominated schedules list if it is not dominated by any schedule in the list. After that, any dominated solutions that are currently in the list are deleted. The value of r increases by 0.05 at the next step and the neighborhood evaluation is performed again. The procedure continues in this way until $r = 1.0$. Then they start with the stable schedule and $r = 0.95$. At each step r decreases by 0.05, and the neighborhood evaluation mechanism is the same. At the end, a list of non-dominated schedules is obtained. The second algorithm of swapping methods (called *bicriterion steepest descent*) uses a bicriterion objective function rather than the convex combination of makespan and the relevant deviation measure. Here again, neighboring solutions are generated with the same method as in *r-grid search*. However, due to the bicriterion objective, it is not possible to identify the "best" neighboring solution. Rather a set of non-dominated solutions is identified. Each solution in this set is a node in a search tree that should be further expanded. The expansion stops when a node dominates all its neighboring solutions. The leaves of this search tree are a list of non-dominated schedules. The initial schedules for this algorithm are taken as the schedules in the list found by *r-grid search* method.

For GA type local search, the initial population is generated by an algorithm, which the authors call as α-ε *search procedure*. Another alternative for the initial population is the output list of *r-grid search* method. At each iteration, a number of chromosomes are installed directly to the next population (asexual reproduction). The rest of the next population is obtained by selecting the most-fit chromosomes. The fitness measure of a given solution *i* is defined as follows:

$$f_i = \frac{\left[I_{MAX} - I_i\right]^k}{\sum_{j \in P}\left[I_{MAX} - I_j\right]^k} \quad where$$

$$I_{MAX} = \sqrt{\left[\beta_1 M(\pi_d)\right]^2 + \left[\beta_1 M(\pi_c)\right]^2}$$

$$I_i = \sqrt{\left[\beta_1 M(\pi_i)\right]^2 + \left[\beta_1 M(\pi_i)\right]^2}$$

β_1 and β_2 are the normalization constants that unify the scale of the makespan and relevant deviation measure; π_d, π_c, and π_i are the Carlier's, the minimum deviation and the ith schedule, respectively; *k* is a constant used for tuning purposes. For chromosome encoding a string of "dummy tails" are used because this type of encoding ensures feasible schedules result after mutations and crossovers.

For testing the performances of these algorithms, they commit two sets of experiments. In the first set, they generate small problems and compare their solutions with the optimal solution obtained by a mixed integer-programming model. In the second set, they measure the effectiveness of their heuristics in large problems. As the result of the first set of experimentation, all heuristics were able to find optimal solutions. As a result of the second set of experiments, they confirmed that stability of the schedules could be improved significantly with little sacrifices in makespan. No significant difference is observed between algorithms but computational burden of all pairwise swapping heuristic is more than that of adjacent pairwise swapping heuristic and genetic algorithm is in between.

Leon *et al.*'s (1994) objective is to find an initial schedule for a job shop system such that expected makespan and expected deviation from the initial makespan are optimized under the machine breakdowns. They use a genetic algorithm to find the best schedule. The authors also employ average slack in the system as a surrogate measure

for expected deviation. Expected makespan is equal to makespan of the initial schedule plus this deviation. The computational experiments indicated that the proposed method is effective in the high processing time variability and machine breakdowns. The uninterested user may skip the following paragraph, which gives the details of this study, without loss of continuity.

Leon *et al.* (1994) study robustness measures and robust scheduling in a job shop environment. Their aim is to construct a robust initial schedule. Given a response policy, an *a priori* off-line schedule that maintains high performance in the presence of disruptions is said to be robust. They assume a "right-shift" response policy. In their model, the disruptions are machine failures. The times to failure and repair distributions are assumed to be known. The shop floor performance measure is taken as makespan. The robustness measure for a given schedule is represented as a convex combination of expected makespan of the realized schedule and expected deviation from the original deterministic makespan. That is the objective is to find a schedule S, that minimizes $R(S) = r.E[M(S)] + (1-r).E[\delta(S)]$, where r is a real number between 0 and 1, $M(S)$ is realized makespan of schedule S and $\delta(S)$ is deviation of realized makespan from anticipated deterministic makespan. The authors calculate $E[\delta(S)]$ analytically for the single machine case with a single machine failure. Note that $E[M(S)]$ can be calculated easily if $E[\delta(S)]$ is known, thus $R(S)$ can be calculated. However, in a job shop environment in the face of multiple machine failures, calculation of $\delta(S)$ analytically is intractable. Under the insight they gain for the single machine single disruption case, they propose several surrogate measures for $E[\delta(S)]$ for job shop systems. With a correlation study, they determine the best surrogate measure for $E[\delta(S)]$ and hence $R(S)$. Then, they propose a genetic algorithm to minimize this surrogate measure of $R(S)$. They conduct two sets of experiments to test the performance of their algorithm under disruptions. In the first set of experiments, they examine the effects of machine breakdowns whereas under the second set of experiments, the effects of processing time variability are examined. The results indicate that under machine breakdowns, robust schedules outperform the schedules that are generated to minimize makespan only. As a

result of the second set of experiments, they find out that robust schedules are better than makespan schedules only processing time variability is relatively high.

Daniels and Kouvelis's (1995) objective is to find a robust job sequence for a single machine under processing time variability such that performance measure degradation under the worst possible scenario is the least. They use a branch-and-bound method and two approximating heuristics to find such a schedule. The computational experiments indicated that the proposed heuristics are efficient and the sequences found by their algorithms are more robust than the sequences that are found by priority dispatching rules that only consider system performance. Following two paragraphs give the details of this paper.

Daniels and Kouvelis (1995) generate initial robust schedules to hedge against processing time variability in a single machine environment. They argue that either deterministic or stochastic models developed up-to-date fail to accommodate several factors that are relevant in many scheduling environments properly. Deterministic models assume all scheduling parameters can be specified precisely in advance of scheduling, allowing the outcome of any scheduling decision to be determined exactly, which is generally not the case in practice. On the other hand, stochastic models adopt a probabilistic viewpoint and treat job attributes as independent random variables with given distributions. In practice, distributions can only be specified imprecisely. They state that a few factors determine the uncertainty of attributes of many jobs, so imposed distributional independence assumptions for the sake of mathematical tractability are not justified. Hence, they propose a scenario-based representation and analysis of uncertainty rather than using stochastic models. They use second measure and policy of section 3.1.2 for robustness. Recall that this policy finds the schedule whose performance degradation under its worst case scenario is the least.

They study a single machine problem where the performance measure is total flow time, and the source of uncertainty is processing time variability. They formulate mathematical programs (called as *absolute deviation robust scheduling problem* ADRSP and *relative deviation robust scheduling* RDRSP) whose solutions gives the

robust schedules, when the number of scenarios is finite (Policy 3 of Section 3.1.2). ADRSP and RDRSP differ from one another only in the definition of robustness. The first one measures the deviations from optimal absolutely, whereas the other measures relatively (similar to percentage deviation). They prove that this problem is NP hard. Besides, in practice, the number of scenarios is infinite, because possible processing time values for jobs are given as ranges, for example processing time of job 1, $P_1 \in [a, b]$. Hence, this problem is expected to be more difficult. However, they prove that properly selected finite set of scenarios are enough to determine the worst-case absolute deviation of a given sequence and construct a procedure that does the worst case evaluation in polynomial time. They develop a branch and bound algorithm and two $O(n \log n)$ surrogate relaxation heuristics that utilize this procedure to generate a robust sequence. They test the efficiency of these algorithms. As expected, computational burden of branch and bound algorithm, which solves the problem to optimality, grows rapidly with problem size. Still the number of sequences evaluated is small as compared to total number of sequences, $n!$, where n is the number of jobs. The heuristics require far less computational time because they evaluate much less permutations of jobs to determine approximate robust sequences. Their computational burden grows only modestly as problem size increases. Moreover, they closely approximate the optimal absolute deviations. They compare their solutions to SEPT (shortest expected processing time) solutions, which is used in practice to generate optimal sequence of jobs. They observe SEPT performs poorly in terms of robustness.

Similar to the study of Wu *et al.* (1993), Mehta and Uzsoy (1998, 1999) work on generating a stable schedule. Unlike the previous study, they use the maximum lateness (instead of makespan) as a performance measure and apply their approach both to the single machine and the job shop systems. Their major difference from the previous study is that Mehta and Uzsoy aim at finding an initial stable schedule before the disruptions as compared to Wu *et al.* (1993) find the new schedule after disruptions. They generate the stable initial schedules by inserting idle times to the schedules that optimizes sysytem performance. What follows is the details of these studies.

Mehta and Uzsoy (1998, 1999) generate initial stable schedules under random machine breakdowns. They call initial schedules as *predictable* schedules. Like Wu *et al*. (1993), Mehta and Uzsoy argue that a good predictable schedule should not only have a good shop floor performance, but at the same time should take *predictability* (i.e. stability) into account. That is, deviations from the original schedule should also be at a minimum level, because many other decisions (such as purchasing, tooling, etc.) in shop floor are planned according to this initial schedule. The authors use the term "predictable scheduling" as an alternative approach to what is known as scheduling/rescheduling in the classical scheduling literature. The objective of this approach is to generate an initial schedule such that deviations from initial schedule during execution is minimized while keeping shop floor performance degradation in an acceptable level. The specific problem they study in the first paper is the single machine scheduling problem where jobs have nonzero ready times and random machine breakdowns are present. The time-to-failure and repair duration distributions are assumed to be known *a priori*. In the second paper they study the job shop scheduling problem with random machine breakdowns. In both studies, they use maximum lateness as shop floor performance measure. Unlike Wu *at al*. (1993), they consider the minimization of deviations while generating an initial schedule, not when rescheduling after a breakdown. Deviations from the initial schedule is measured in terms of expected absolute deviation of job completion times. The authors offer a two stage approach: in the first stage a job sequence is determined to minimize the maximum lateness. They use Carlier's algorithm and shifting bottleneck algorithm for the single machine case and the job shop case, respectively. In the second stage, they insert some idle times in this sequence. The amount of idle times are determined in such a way that they are large enough to provide enough protection against machine breakdowns but they are small enough so that maximum lateness does not increase much. Since minimizing expected absolute job completion times deviations explicitly is too difficult, they develop five alternative surrogate measures that are correlated with the original objective. After a detailed correlation analysis, they choose the best surrogate measure and use in the rest of the paper. The authors propose basically two

algorithms to determine the amount of idle times. The first one is called as *optimized surrogate measure heuristic* (OSMH). In this heuristic, the amount of idle times are determined as expected machine repair durations during the processing of an operation. The second algorithm is a linear-program, where increase in maximum lateness is restricted by adding it as a constraint to the program. The solution of this program gives start time of each operation. This linear program (LPH) yields the same result as OSMH if maximum lateness degradation level is not restricted. In the single machine case, they also develop a tabu-search algorithm to determine the effects of changing job sequence but found that changing the job sequence does not significantly improve the performance. In the computation experiments, they compare OSMH and LPH schedules with classical maximum lateness algorithms and conclude that predictability is can be easily improved while slightly increasing maximum lateness.

Similar to the study of Mehta and Uzsoy (1999), O'Donovan *et al.* (1999) work on generating stable schedules. Unlike the previous study, they use the total tardiness (instead of maximum lateness) as a performance measure and beside generating a stable initial schedule, they also work on rescheduling policies in response to machine breakdowns. Another difference from the previous study is that O'Donovan *et al.* (1999) also consider *sensitive jobs*. They develop a model where machine breakdowns affects the processing times of jobs (the effect decreases gradually as time passes and degree of the effect varies from job to job) and study scheduling/rescheduling policies on this model. The details of this paper are as follows.

O'Donovan *et al.* (1999) examine the scheduling/rescheduling policy using stability and efficiency measures in a single machine environment. The schedule efficiency is measured by total tardiness. The stability is measured by absolute completion time deviations from the initial schedule. The system under study has nonzero job ready times and random machine breakdowns. This study is similar to the one by Mehta and Uzsoy (1999) except that total tardiness is used instead of maximum lateness as the system performance measure. They consider ATC(1) and ATC(1) + OSMH for initial schedule generation. According to the ATC(1) heuristic (Rachamadugu and Morton, 1982), whenever the machine becomes free, the job with

the highest priority index is scheduled next, where the priority index of job i at time t is given by

$$\pi_i(t) = (1/p_i)[\exp(-\max\{d_i - (t + p_i), 0\}/kp_{av})].$$

Here p_i is the processing time of job i, d_i is the due date of job i, p_{av} is the average processing time of all jobs (computed at the start of the procedure), and k is a look-ahead parameter that governs how fast the job's priority increases as its slack decreases. ATC(1) + OSMH heuristic generates initial sequence with ATC(1) and additional idle times are inserted according to the OSMH heuristic described previously (Mehta and Uzsoy, 1999). Rescheduling alternatives are ATC(1), ATC(2) and right-shift scheduling. Instead of using due date as in ATC(1), ATC(2) calculates the slack of each job based on its predicted completion time, $C_i(S_p)$. The priority index of job i is given by

$$\pi_i(t) = (1/p_i)[\exp(-\max\{C_i(S_p) - (t + p_i), 0\}/kp_{av})].$$

Experimental results indicate that ATC(1) + OSMH for scheduling and ATC(2) for rescheduling is the best for stability. The authors also consider *sensitive jobs*. They argue that in some production environments, jobs are sensitive to disturbances that have just occurred, and the degree of sensitivity differs from job to job (the impact of a disruption is not felt equally by all jobs). In their proposed model, they estimate expected increase in job processing time as

$$E[p_i^*] = \tau_i p_i[\exp(-\alpha E[T])],$$

where $E[T]$ is the expected time between breakdowns, α is the recovery rate of machine, and τ_i is the impact factor of job i. In this model, they again study scheduling and rescheduling alternatives and propose ATC(1) + OSMH for scheduling and Smart ATC(2) for rescheduling. Smart ATC(2) is similar to ATC(2), but it uses estimated affected processing time instead of processing time for job i.

Finally, Wu *et al.* (1999) proposes a quasi-online method to generate robust schedules. They study the job shop scheduling problem under disruptions (processing time variability). Their approach (called as *PFSL* - Process First Schedule Later - approach) combines the global viewpoint of off-line methods and adaptability of on-

line methods. A small number of major scheduling decisions are made by a branch-and-bound algorithm and the rest of the schedule is filled dynamically by the ATC heuristic during execution. Their computational experiments indicate that their quasi-online method is superior to traditional off-line and on-line methods in terms of schedule robustness. The following paragraph gives the details of this study.

Wu *et al.* (1999) propose a graph-theoretic decomposition to the job shop scheduling problem to achieve schedule robustness. They use the *schedule robustness* in the sense of adaptability to system disturbances. They state that off-line methods to scheduling assume perfect information, hence the generated schedules are not adaptable to external disturbances. On the other hand, on-line methods are very adaptive due to their dynamic nature, but lack the view in a global perspective. In this study, the authors combine good properties of both approaches and propose a quasi-online method for the job shop scheduling problem. The robustness measure is expected average weighted tardiness. They use graph representation of this problem, in which conjunctive arcs represent precedence constraints and disjunctive arcs join the operations competing for the same resource. They propose a branch and bound algorithm that assigns directions some of these disjunctive arcs, and hence changing them into conjunctive arcs, and effectively give some of the scheduling decisions. The remaining scheduling decisions are made dynamically by applying ATC heuristic. In their paper, this approach is called as *process first schedule later (PFSL)* scheme. They compare their approach with the off-line algorithm (IATC) and online algorithm (ATC). Their computational experiments show that *PFSL* scheme yields more robust performance under a wide range of disturbances (various levels of processing time variability) as compared to traditional off-line and on-line methods.

In reactive scheduling literature, several other authors develop schedules in the face of disruptions without considering disruption in the decision making phase. Here we review some of the recent studies to give the reader a flavor of this line of research. The interested reader is referred to Sabuncuoglu and Bayiz (2000) for a broader literature review.

Church and Uzsoy (1992) analyze the performance of event-driven scheduling in a single machine environment with dynamic job arrivals. They classify the events that change the system state into two categories: 1) the events that require immediate response (exceptions) and 2) the events that can be ignored until the next rescheduling point. The schedule is revised periodically but scheduling is also triggered when an exception occurs. In their model, the exceptions are arrivals of jobs with tight due dates. For each job *i* arriving between times $(k-1)T$ and T, they calculate the slack

$$s_i = d_i - r_i$$

where d_i is the due-date r_i is the ready time of job *i* and T is the period length. If this value is smaller than a constant w (window length), then the arrival of job *i* is considered as an exception and a scheduling decision is triggered. A schedule is also generated at the beginning of each period (at the times kT). The authors use EDD dispatching rule to generate schedules at each revision. The performance measure is maximum lateness. Their computational experiments indicated that benefits of extra scheduling diminish rapidly. They conclude a well-designed event-driven policy can achieve good system performances with less computational burden as compared to scheduling in response to every event that change the system state.

Akturk and Gorgulu (1999) study on the rescheduling of operations in a modified flow shop environment in response to a machine breakdown. In a modified flow shop, jobs can enter the system at one of the several machines, can progress through the system by a limited number of paths and can exit the system on one of the several machines. Hence, it falls somewhere between a flow shop and a job shop. The authors assume that an initial schedule is available and it is followed until a single machine breakdown occurs. In response to the machine breakdown, they reschedule the operations to match up with the initial schedule at a point in the future. In the first stage, they determine a match-up point for each machine. Then the authors decompose the rescheduling problem into three parts: 1) the scheduling of the down machine, 2) the scheduling of the machines in the upward direction of the down machine and 3) the scheduling of the machines in the downward direction of the down machine. If a resulting schedule is not feasible, then the match-up point is changed to enlarge the set

of jobs that are rescheduled. Their experimental results indicate that the proposed algorithm is very effective in terms of schedule efficiency, computational times and schedule stability.

In another study, Sabuncuoglu and Karabuk (1999) investigate the scheduling/rescheduling problem in an FMS environment. The authors propose a filtered beam search heuristic for FMS environment. For several reactive scheduling policies in response to machine breakdowns and processing time variability, the authors compare off-line and on-line scheduling algorithms. The performance of the system is measured in terms of makespan and mean tardiness. Their computational experiments indicate that the proposed off-line algorithm performs better than on-line machine and AGV scheduling rules, under all experimental conditions for the makespan, mean flow time and mean tardiness criteria. They also show that it is not always beneficial to reschedule the operations in response to every unexpected event. They conclude that the periodic response with an appropriate period length can be effective to cope with the interruptions.

Sabuncuoglu and Bayiz (2000) study the reactive scheduling problem in a job shop environment. The authors measure the effect of shop floor configuration (system size and load allocation) on the performance of the scheduling methods (off-line and on-line). Their performance criteria are makespan and mean tardiness. In the first part of the study, they compare a beam search based heuristic to other well-known algorithms including Lawrance (1984), Adams, Balas, and Zawack. (1988) and Applegate and Cook (1990). In the second part, they study on different reactive policies such as partial scheduling versus full scheduling, etc. Their computational experiments indicated that beam search is quite promising for the job shop problem and partial off-line scheduling can be a very practical tool in a highly dynamic and stochastic environment.

Table 4. Classification of Studies

Author	Environment			Schedule Generation				
	Shop Floor	Sta./Dyn.	Stoch./Det.	Method	Objective	When to	How to	
							Scheme	Response
Wu *et al.* (1993)	Single Machine	Static	Stochastic (Machine breakdown)	GA Pairwise swapping methods	Minimize deviation between start times or between sequences (stability) Minimize makespan	Continuous (Machine Breakdowns with BR=1)	Off-line	Reschedule (same method)
Leon *et al.* (1994)	Job Shop	Static	Stochastic (Machine breakdown, processing time variability)	GA	Minimize expected makespan and expected deviation from original makespan using surrogate measures (robustness)	Continuous (Machine Breakdowns with BR=1)	Off-line	Right Shift
Daniels *et al.* (1995)	Single Machine	Static	Stochastic (Processing time variability)	B&B, other heuristics	Minimize absolute worst case total flow time difference (robustness)	Periodic	Off-line	Do nothing (left or right shift)
Mehta and Uzsoy (1998)	Job Shop	Static	Stochastic (machine breakdown)	OSMH/LP	Minimize deviations between completion times while keeping L_{MAX} low using surrogate measures(stability)	Continuous (Machine Breakdowns with BR=1)	Off-line	Right Shift
Mehta and Uzsoy (1999)	Single Machine	Static with nonzero ready times	Stochastic (Machine breakdown)	OSMH/LP	Minimize deviations between completion times while keeping L_{MAX} low using surrogate measures(stability)	Continuous (Machine Breakdowns with BR=1)	Off-line	Right Shift

Table 4. (Cont'd)

O'Donovan *et al.* (1999)	Single Machine	Static with nonzero ready times	Stochastic (Machine breakdown, processing time variability)	OSMH and ATC derivatives in combination	Minimize deviations between completion times while keeping total tardiness low (stability)	Continuous (Machine Breakdowns with BR=1)	Off-line	Right Shift ATC derivatives
Wu *et al.* (1999)	Job Shop	Static	Stochastic (Processing time variability)	B&B/ATC	Minimize expected weighted total tardiness using surrogate measures(robustness)	Periodic	Quasi on-line	Right-Shift
Church and Uzsoy (1992)	Single Machine	Static with nonzero ready times	Deterministic	EDD	Minimize maximum lateness (efficiency)	Event-driven(Periodic + Urgent jobs)	On-line	Reschedule
Akturk and Gorgulu (1999)	Modified Flow Shop	Static	Stochastic (Machine breakdown)	RHSA	Minimize tardiness, earliness (efficiency)	Continuous(Machine Breakdowns with BR=1)	Off-line	Match-up
Sabuncuoglu and Karabuk (1999)	FMS	Static	Stochastic (Machine breakdown, processing time variability)	Filtered beam search	Minimize mean tardiness, mean flow time and makespan (efficiency)	Continuous(Machine Breakdowns with BR=1)	Off-line	Reschedule
Sabuncuoglu and Bayiz (2000)	Job Shop	Static	Stochastic (Machine breakdown)	Filtered beam search	Minimize mean tardiness, and makespan (efficiency)	Continuous(Machine Breakdowns with BR=1) Periodic	Off-line	Reschedule

Chapter 4

Observations and Proposed Methods

4.1. Observations

On reviewing the existing literature, the following observations are made:

1. We know that in a static and deterministic environment off-line scheduling is superior to on-line scheduling. In a static and dynamic environment, off-line scheduling is still better, but the difference between the performances of off-line and on-line scheduling is not as large as in the static case (Sabuncuoglu and Karabuk, 1999). The comparison of off-line scheduling and on-line scheduling methods is needed in a dynamic and stochastic environment.
2. It is also known that full scheduling is superior to partial scheduling in a static environment. (Sabuncuoglu and Bayiz, 2000). The relative performance of full scheduling and partial scheduling in a dynamic and stochastic environment is also an open research question.
3. The type of response issue (scheduling/rescheduling, or repairing) has not been thoroughly studied in the literature. Even though there are some studies that focus on rescheduling frequency, their relative weaknesses and strengths are not generally known.
4. When-to-schedule policies (periodic, continuous, or adaptive) needs a further research. Specifically, we do not know the conditions under which each method is better than others.

5. Should practitioners consider robustness or stability or both? What is the trade-off between these performance metrics (stability, robustness, and efficiency)? It should also be investigated.
6. Along the same lines, can one develop a bicriterion approach that considers both the stability and robustness measures simultaneously?
7. As stated in Mehta and Uzsoy (1998), can one develop better surrogate measures for stability (and/or robustness)?
8. As also questioned by Leon *et al.* (1994), can one extend the existing studies for other performance measures?
9. As stated in Daniels and Kouvelis (1995), can one apply the existing concepts to other realistic scheduling environments?

Each of these observations can be identified as a further direction of research. In this study, we decided to focus on development of alternative surrogate measures for robustness and stability (item 7 in the above list). The details of the proposed methods can be found in the next section. Our computational experiments show that the proposed methods are better than the average slack method, which is commonly used as a surrogate measure in the literature. Moreover, as a by-product of our study, we also obtained some partial answers to items 5 and 6 in the above list, as explained later.

4.2. Proposed Methods

From the viewpoint of robustness, what really matters is the performance of the *realized* schedule rather than the performance of the initial schedule generated on a piece of paper beforehand. In terms of stability, a good realized schedule should deviate minimally from the initial (or planned) schedule. In a way, both measures (robustness and stability) are the performance metrics on the realized schedule, which cannot be known in advance. It seems that one can only predict the realized schedule and estimate its performance by using surrogate measures. Most of the studies in the literature employ average slack method to generate robust and/or stable schedules. This method

is developed by Leon *et al.* (1994) to generate robust schedules for the job shop scheduling problem. They use makespan as the performance measure. They estimate the expected realized makespan of a schedule by $E[f(S)] = f(S_0) + E[\boldsymbol{d}(S)]$ where *E[f(S)]* is the expectation of the realized performance measure; $f(S_0)$ is the initial (or planned) performance measure (makespan) of S, and $E[\boldsymbol{d}(S)]$ is the degradation in the performance measure because of the disruptions, which is estimated by using –(average system slack) as a surrogate measure. For each operation the difference between the earliest start time and the latest time at which that operation can start without causing delay is called as the *slack* of that operation. Average system slack is calculated by summing slacks of all operations and dividing that number to the total number of operations. The authors' experimental studies indicate that there is a high correlation between the robustness of a given schedule and the average slack of the schedule.

The use of average slack method to generate stable schedules comprises similar ideas: a stable schedule should have a large average slack value to increase the probabilty that the schedule will absorb the negative affects of a disruption. The experimetal studies indicate that there is a hihg correlation between the stability of a given schedule and the average slack of the schedule.

In this study, we propose two new surrogate measures to estimate the realization of a given schedule for the single machine system with random machine breakdowns. We use an algorithm that is based on tabu search methodology to efficiently evaluate a large number of sequences in search space. We assume that arrival times, processing times and due-dates are known in advance. The distributions of busy time (*f(t)*) and repair duration (*g(t)*) are assumed to be known *a priori.*

4.2.1. The Proposed Scheduling Algorithm

In this study, our objective is to construct robust and stable schedules. The robustness measure is expectation of the realized schedule performance (Measure 1 of Section

2.2.1.1 of Chapter 2). The stability measure that we use is the expected absolute deviation of job completion times (Measure 1 of Section 2.2.2 of Chapter 2).

In the proposed algorithm we start with an initial job sequence and use tabu search to scan the solution space efficiently. At each iteration, we assess the quality of a number of alternative sequences and adopt the best sequence as the current solution. The details of the tabu search are presented in the next section. We use two surrogate measures to assess the quality of a sequence. Instead of using an indirect surrogate measure such as average system slack, we quickly estimate the performance of a candidate sequence by two different methods. We use the values of the estimated performance as a surrogate measure for robustness. Similarly, the sum of absolute deviations of the job completion times in the initial sequence from the job completion times in the estimated realization is used as a surrogate measure for stability. As explained later in Chapter 5, we also include the average slack method and the classical approach (which does not consider robustness or stability) to asssess the quality of a given schedule as benchmarks in our experimental study. Note that the average slack method for robustness is devised for the job shop scheduling problem and the makespan criterion. We adopted the method to the single machine scheduling problem by calculating the slacks of the jobs (rather than the operations) and taking the average. We also included total tardiness and total flow time criteria to our study. The reason why we use this modified method (and accept the risk of deteriorating the performance of the original method) as a benchmark is that there is no robustness procedures spesifically developed for the single machine problem in the literature.

The details of the estimation methods are presented in Section 4.2.3. Figure 4 illustrates the basic idea behind the proposed algorithm.

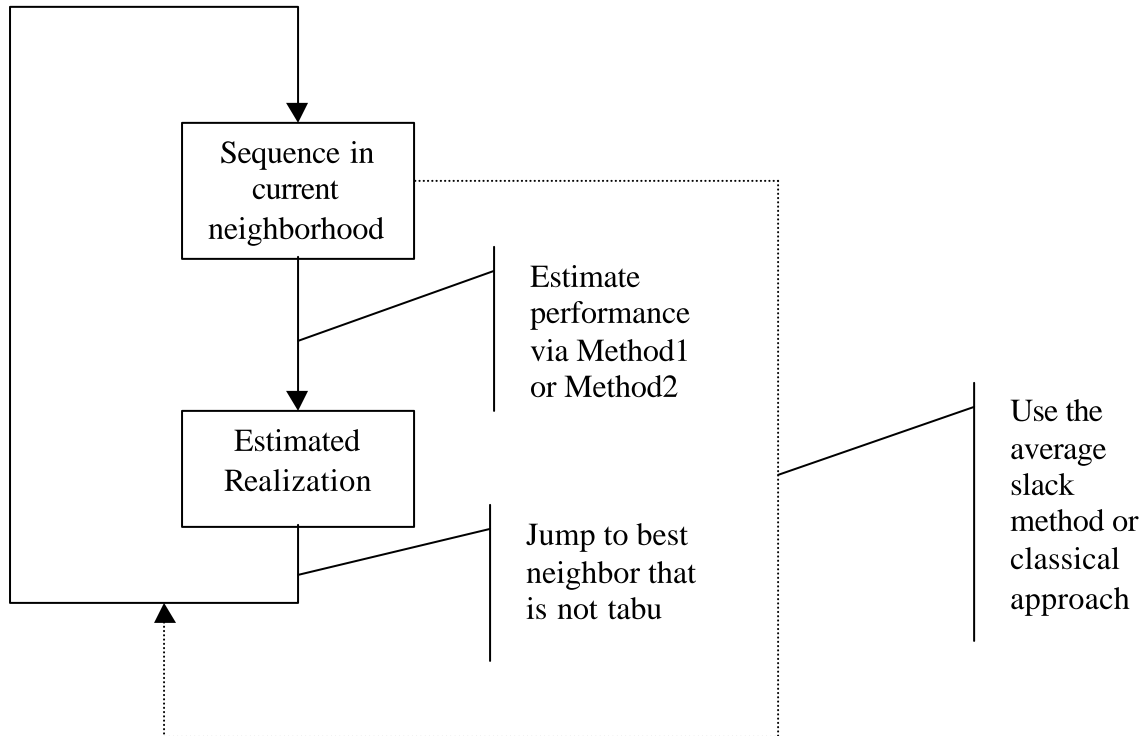


Figure 4. Illustration of Proposed Algorithm

4.2.2. Tabu Search

Tabu Search (TS) was developed by Fred Glover as a search technique for solving a wide variety of combinatorial optimization problems. TS has been applied in many areas including scheduling, layout design and location, allocation, telecommunications, production, inventory and investment, routing, and graph optimization (Glover and Laguna, 1997).

Tabu Search method was partly motivated by the observation that human behavior operates with a random element that leads to different courses of actions under similar circumstances. The resulting tendency to deviate from a planned course might lead to errors but can also be source of gain. TS method operates in this way with the exception that new courses are not chosen randomly. Instead, TS assumes that there is

no point in accepting a new (poor) solution unless it is to avoid a path already investigated. This insures new regions of a problems solution space are investigated with the goal of avoiding local minima and ultimately finding the global minimum.

The search mechanism of TS is very similar to that of *steepest ascend* method: at each step the neighborhood of the current solution is generated and the best one among them is adopted as the new current solution. The neighborhood consists of all the solutions that can be generated from the current one via a single *move* (pairwise swapping, inserting, etc.). To prevent immediate backtracking, most recently executed moves (i.e. *tabus)* are kept out of consideration for a certain number of iterations. This number is called as *tabu tenure.* A move can be executed even if it is in *tabu list* if the resulting solution is better than the best one found so far. This is known as *aspiration*.

Laguna, Barnes, and Glover (1991) discuss the use of three tabu search strategies for the approximate solution of a single machine scheduling problem with sequence dependent set-up costs. The objective is to minimize the sum of set-up costs and linear delay penalties for N jobs, all arriving at time zero. They first consider a TS method that use the common approach of making swaps to move from one trial solution to another. Next, they consider the use of insert moves. Finally, they construct a TS method that employs both swap and insert moves. Computational experiments show that there is an advantage of using this hybrid approach, but the improvement is not substantial. In addition, our primary goal is to find out how different methods of evaluating neighbourhood (using average slack, Method 1, and Method 2) compare with each other. Therefore, in our implementation we stick to the common approach of using swap moves, whose implementation is easier.

As stated before, in our implementation, we use swap moves to generate a schedule in the neighbourhood of the current schedule, that is, the neighborhood of a solution is generated by all pairwise swaps of jobs. Therefore, the neighbourhood of a schedule with n jobs consists of $\frac{n(n-1)}{2}$ schedules. The performances of the sequences in the neighborhood are evaluated by our proposed surrogate measures that we calculate via Method 1 or Method 2, which are discussed in the next section. The

neighborhood is then sorted according to their performances. Tabu list consists of triples of numbers (x, y, z), where x is the job identification number (id), y is the position in the sequence and z is the remaining tabu tenure. If (x, y, z) is in the tabu list, job x cannot move to position y during next z iterations. A swap is identified as tabu if one of the corresponding moves are in tabu list. For example suppose our current sequence is "*a b c d e…*" The swap of job a with job e is tabu if one of (a, 5, .) or (e, 1, .) is in the tabu list. Beginning with the best sequence in the neighborhood, corresponding swaps are tested. If the swap is not tabu, then it is executed. If the swap is tabu but the resulting sequence performs better than the best sequence found so far, it is still executed (this is known as aspiration). Otherwise, the swap that gives the next best solution in the neighborhood is tested. Continuing in this way, if all possible swaps are found tabu, the swap that gives the best one in the neighborhood is executed. After execution of a swap, corresponding two moves are added into the tabu list, if they are not already in. For example, assume that the current candidate is "*a b c d e …* ". If swap of job a with job c is executed, then (a, 1, .) and (c, 3, .) entries are added to the tabu list. The sequence resulting after this swap is adopted as the new current solution. This solution is compared with the current best solution found so far. If it is better, the best solution found so far is set to the current solution. Tabu tenures in the tabu list are decreased, and the moves with 0 remaining tabu tenures are deleted. If the stopping criterion is not satisfied, the next iteration is executed.

4.2.3. Proposed Neighbourhood Evaluation Methods

4.2.3.1. Method 1

In our opinion, the well-known and commonly used average slack method fails to incorporate the information that can be inferred from the probability distributions of busy time and repair-time. In contrast, the proposed method, Method 1 estimates the performance of the realized schedule using the relevant distributions themselves. This is explained next.

The essence of Method 1 is to quickly estimate the realized schedule that corresponds to a given sequence. The related performance measure of this estimated realization is then used to measure the quality of a given sequence in the search space. Assume that the machine fails after every busy time period of length ***1****L* + *(1-****1****)U*, where ***1*** is a real number between 0 and 1, and L and U are points on the left and right tails of *f(t)*, respectively. As shown in Figure 5, the probability that a machine breakdown will occur between L and U is 0.95 (i.e, $\alpha = 0.05$). We further assume that the repair activity lasts *E[g(t)]* time units (i.e. the expectation of the repair-time distribution).

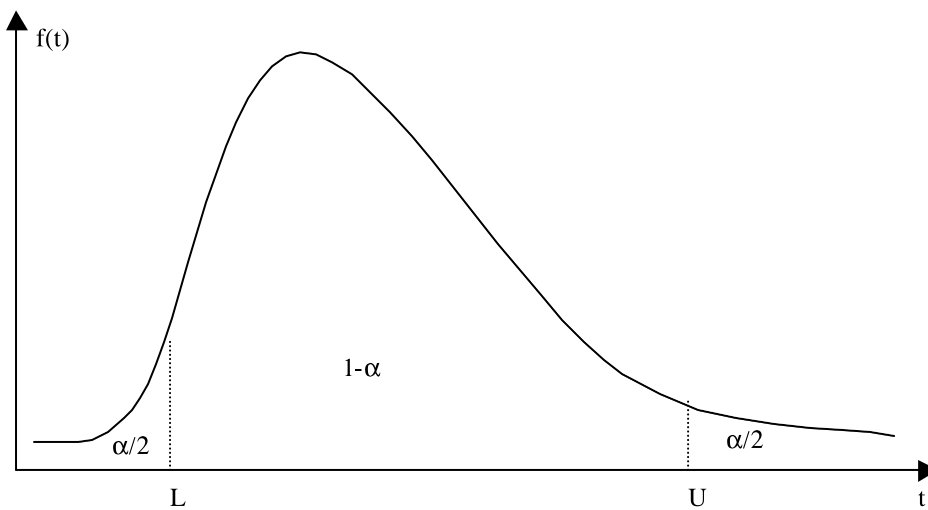


Figure 5: Parameters of Method 1

The parameter ***1*** is to be determined from the correlation study, in which a number of pilot schedules are to be estimated using ***1*** values of 0.2, 0.4, 0.6, and 0.8 along with the alternative of using *E[f(t)]* instead of ***1****L* + *(1-****1****)U* as busy time period. Figure 6 illustrates the estimation of realized schedule.

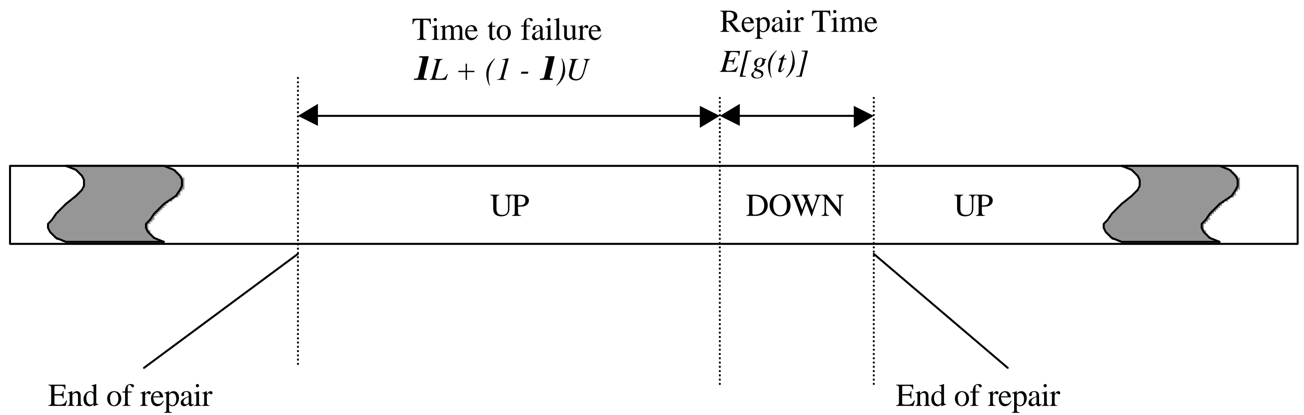


Figure 6. Estimation of Realization under Method 1

4.2.3.2. Method 2

Like Method 1, Method 2 estimates the realization of a given schedule. This approach, however, assumes that there is only one machine failure during any scheduling period. Because of this restrictive assumption, it is better to use this approach with a continuous rescheduling scheme. Given a sequence of n jobs, S, and busy time distribution, $f(t)$, the expected realized performance is estimated in three steps:

1. Calculate the probability that the machine fails during the processing of ith job. As seen in Figure 7, a_i is the probability that the machine fails during the processing of ith job.
2. Determine the performance of the sequence assuming that machine actually fails during the processing of job i. Use $E[g(t)]$ as the repair duration. Let f_i be the value of this performance measure.
3. Calculate the estimated realized performance measure of the sequence S as $\sum_{i=1}^{n} a_i f_i$.

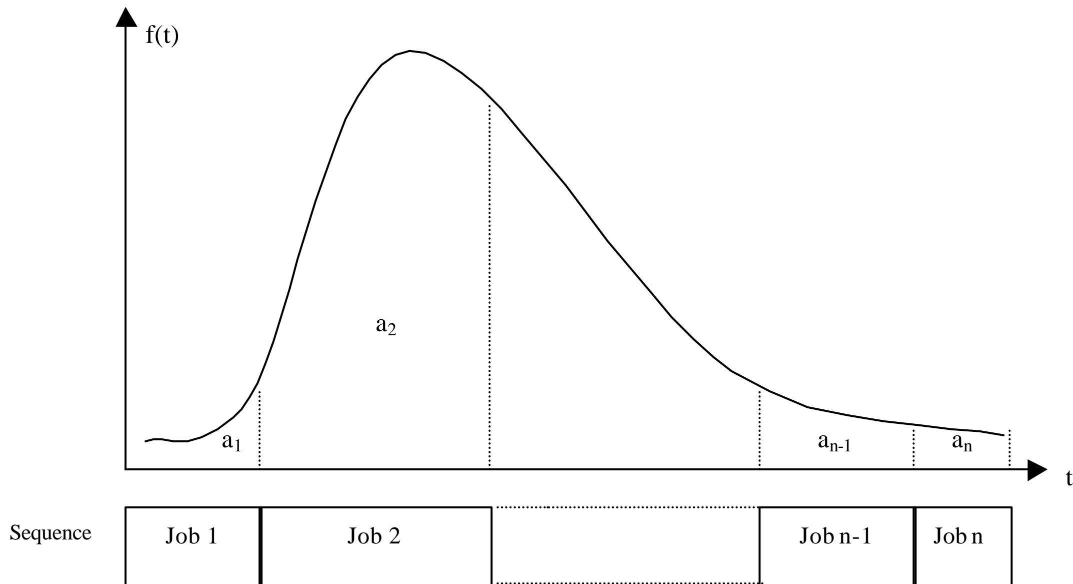


Figure 7. Parameters of Method 2

4.2.4. A Numerial Example

In this section, we give a hypothetical example to clarify the mechanics of the proposed scheduling algorithm.

Assume that we want to generate a robust schedule for the single machine problem with 3 jobs, whose arrival times, processing times, and due-dates are given in Table 5.

Table 5. Job Parameters for the Numerical Example

Job	Arrival Time (hr)	Processing Time (hr)	Due Date (hr)
J1	5	1	6
J2	2	1	5
J3	1	1	4

Assume that busy time distribution of the machine is Uniform(0, 3) and repair duration distribution is Uniform(0, 2) for the sake of simplicity. Let total tardiness

criterion be the performance measure. Thus, we want to generate such a schedule that expected total tardiness of its realization is minimized.

Suppose our algorithm begins with the initial schedule “J1-J2-J3”. The algorithm first generates the neighbourhood of this schedule by all-pairwise swapping. Hence, there are $\frac{3x2}{2} = 3$ schedules in the neighbourhood of the schedule “J1-J2-J3”. These are “J2-J1-J3”, “J3-J2-J1”, and “J1-J3-J2”. Let us call them as S1, S2, and S3 , respectively. Next the algorithm evaluate the quality of S1, S2, and S3 by using the appropriate surrogate measure according to the method in use:

Classical Approach: If the classical approach is in use, the algorithm calculates the initial total tardiness of each candidate schedule (i.e., S1, S2, and S3). The initial schedules are given in Figure 9-a. The total tardiness values of these initial schedules can be found in the second column of Table 6.

Average Slack Method: As explained before in section 4.2, this method use (initial tardiness – average system slack) as the surrogate measure for robustness. Third column of Table 6 gives the values of this measure. The calculation of average system slack can be found in Table 7.

Method 1: Method 1 calculates the total tardiness of realized versions of each candidate schedule (i.e., S1’, S2’, and S3’). The *L* parameter of Method 1 is 0.075 and the value of *U* is 2.925 if we take $\alpha = 0.05$ (See Figure 8). Assume that the value of λ is 0.5. Then, as explained in Section 4.2.3.1, when estimating realization according to Method 1, the algorithm inserts $E[g(t)] = 1$ hour of idle time as repair duration after every 0.5*0.075 + 0.5*2.925 = 1.5 hours of busy time period. These estimated realizations can be seen in Figure 9-b. Fourth column of Table 6 lists the total tardiness values of these realizations.

Method 2: As explained in Section 4.2.3.2, Method 2 assumes a single machine breakdown and calculates the expected total tardiness of the realization of each candidate schedule. For each candidate, the probability that the machine fails during the processing of any job (J1, J2, or J3) is 1/3. Method 2 first calculates the total tardiness of the realized schedule assuming that single machine failure is during the processing of

J1, and then J2, and finally J3. These total tardiness values can be found in Table 8. The weighted average of these values (weights being the corresponding probabilities, all 1/3 in this case) gives us the surrogate measure of Method 2. The fifth column of Table 6 presents the values of this surrogate measure. Figure 9-c illustrates the above calculation for candidate S1 as an example.

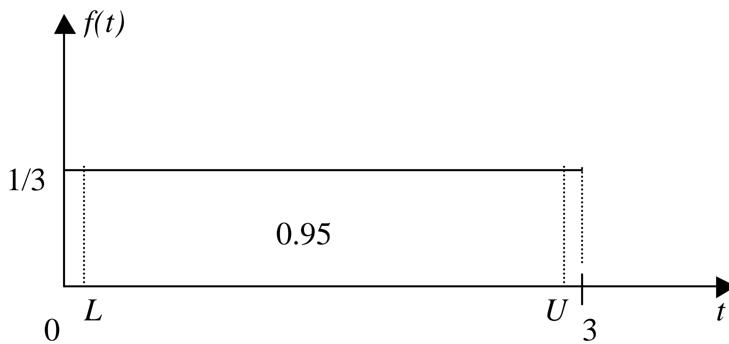
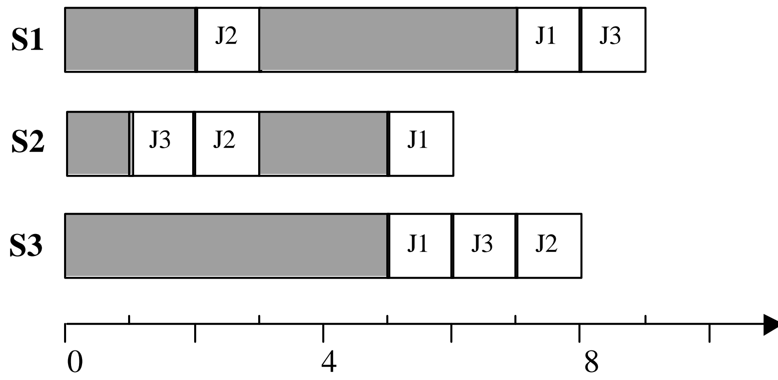
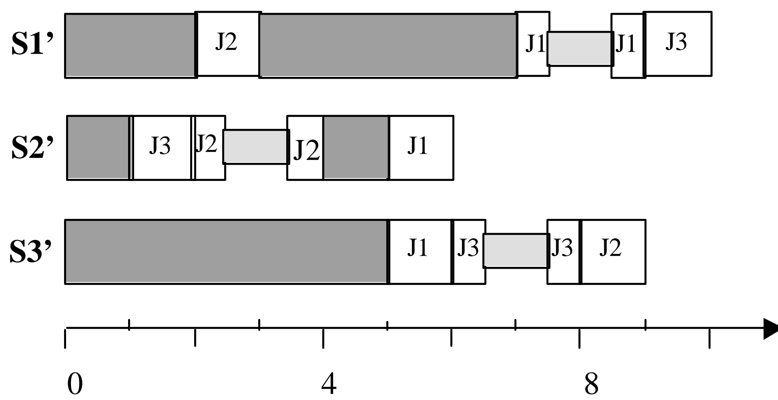


Figure 8. L and U Parameters of Method 1



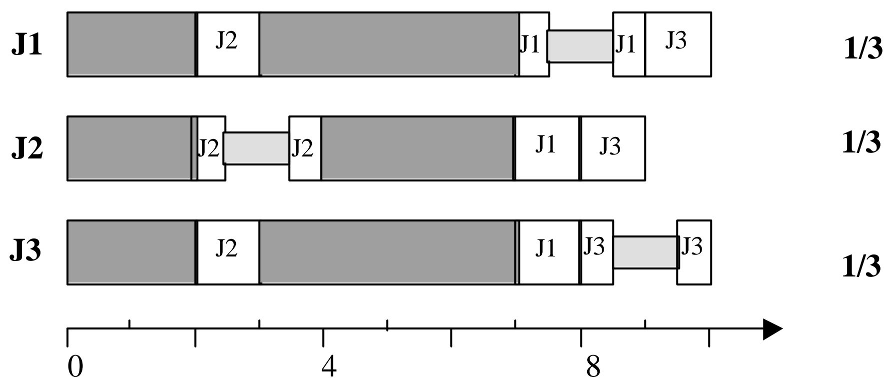
a) Candidate schedules S1, S2, and S3



b) Estimated Realization (S1', S2', and S3') under Method 1

M/C Fails during the Processing of …

Probability of that failure



c) Calculation of expectation for S1 under Method 2

Figure 9. Illustration for Numerical Example

Table 6. Evaluation of Neighbourhood

Schedule in the neighbourhood	Evaluated Candidate Qualities			
	Classical Approach	Average Slack Method	Method 1	Method 2
J2-J1-J3	3	2.33	4	4
J3-J2-J1	0	-1.33	0	0.33
J1-J3-J2	6	6	8	7

Table 7. Calculation of Average System Slack

Job	S1			S2			S3		
	Earliest	Latest	Slack	Earliest	Latest	Slack	Earliest	Latest	Slack
J1	5	5	0	5	5	0	5	5	0
J2	2	4	2	2	4	2	7	7	0
J3	6	6	0	1	3	2	6	6	0
	Average Slack		0.67	Average Slack		1.33	Average Slack		0

Table 8. Calculation of Robustness Measure of Method 2

Schedule	Total Tardiness of Realization If Machine Fail Fails during Processing of …			Expectation
	Job1	Job2	Job3	
S1	5	3	4	4
S2	1	0	0	0.33
S3	9	4	8	7

Next, the algorithm selects the best schedule in the neighbourhood. As seen in Table 6, S2 (J3-J2-J1) happens to be the best solution in the neighbourhood for all evaluation methods. Therefore, S2 is chosen as the new current schedule. This move involves the swap of J3 with J1. Assume that tabu tenure is 12. Then, (J1, 1, 12) and (J3, 3, 12) entries are added to the tabu list (which was empty). The algorithm continues iterating like this until the stopping criterion is satisfied.

Chapter 5

Experimental Study

We have conducted extensive computational experiments to tune up and evaluate our heuristic algorithm. We use an experimental design, whose details are given in the following section, that involves a broad range of test problems. Then, we set the parameters of our tabu search algorithm. Finally, we compare the proposed methods with the other commonly used approaches. We assume a periodic scheduling scheme and type of response is to right shift the remaining jobs (Response BR1 of Table 2), unless otherwise stated. The scheduling program is coded in the C language, which reads the problem parameters from an input file and generates desired schedules. The program (schedule.c) can be found in appendix.

5.1. Experimental Environment

5.1.1. Problem Parameters

In this section, we present the data generation scheme, which is previously proposed by Mehta and Uzsoy (1999), to create test problems.

Number of Jobs (*n*): There are five levels for number of jobs (n = 10, 30, 50, 70, 90). The number of jobs designates the size of the problem. As number of jobs

increases, computational burden, and hence time needed to find the optimal solution, increases.

Processing Time (p_i): Job processing times are generated from discrete uniform distributions. There are two such distributions used – Uniform (1, 11) and Uniform (4, 8), which are referred to as P1 and P2, respectively. P1 and P2 schemes have the same mean, but the variance of P1 is higher than that of P2.

Arrival times (a_i): Job arrival times are generated from a discrete uniform distribution between 0 and $\alpha nE[p_i]$, where $E[p_i]$ is the expected job processing time (=6 time units). Therefore $nE[p_i]$ is the expected makespan of the schedule. A low level for α makes the jobs arrive over a shorter time horizon. Note that if $\alpha = 0$ means all jobs are ready at time 0.

Job due dates (d_i): Job due dates are generated as $d_i = a_i + \gamma p_i$, where γ is generated from a continuous uniform distribution, between a and b. Different a and b levels determines the tightness and the range of the due dates. Four levels of (a, b) are considered as shown in Table 9. D1 and D2 have tighter mean due dates than D3 and D4 have. The range of due dates are higher for D1 and D4 and lower for D2 and D3.

Note that we have 200 possible combinations from the above parameters for test problems (Table 9).

Table 9. Test Problem Parameters

Parameter	Values used in experimentation	Total Values
Number of jobs (n)	10, 30, 50, 70, 90	5
Processing times (p_i)	Uniform[1, 11]) (P1) Uniform[4, 8] (P2)	2
Arrival times (a_i)	a_i = Uniform (0, a_{max}) where $a_{max} = \boldsymbol{a}E[C_{max}]$ $\boldsymbol{a}$ = 0.25, 0.5, 0.75, 1.25, 1.75	5
Job due date (d_i)	$d_i = a_i + \boldsymbol{g}p_i$ where $\boldsymbol{g}$= Uniform [a, b] where values of (a, b) are taken as (-1, 3) (D1) (0, 2) (D2) (2, 4) (D3) (1, 5) (D4)	4

5.1.2. Breakdown Parameters

The scheme for machine breakdown generation is given in Table 10. In the absence of real data, Law and Kelton (1991) recommends the Gamma distribution as a busy time distribution with a shape parameter of 0.7, and a scale parameter to be specified. The authors also state that the Gamma distribution with a shape parameter of 1.4 is appropriate for down time distribution. The scale parameter of the busy time distribution is arranged so that the mean is $\boldsymbol{q}.E[p_i]$. We consider $\boldsymbol{q}$ values of 10 and 3 (Mehta and Uzsoy (1999) use the same scheme with the exception that the distribution is exponential instead of Gamma). The scale parameter of the down time distribution is arranged so that the mean is $\boldsymbol{b}.E[p_i]$. We consider $\boldsymbol{b}$ values of 1.5 and 0.5. Consequently our experimental design consists of 200 problem combinations (Table 9) x 4 breakdown combinations (Table 10) = 800 problem *classes* as a whole.

Table 10. Breakdown Parameters

Parameter	Values used in experimentation	Total Values
Busy time	Gamma distribution with a shape parameter of 0.7 and mean of $\theta.E[p_I]$ θ= 10.0 (Long busy time) (B1, B2) θ= 3.0 (Short busy time) (B3, B4)	2
Down time	Gamma distribution with a shape parameter of 1.4 and mean of $\beta.E[p_I]$ β = 1.5 (Long repair time) (B1, B3) β.= 0.5 (Short repair time) (B2, B4)	2

5.2. Fine Tuning Algorithm Parameters

We conduct pilot runs to determine the levels of the following parameters of the scheduling algorithm:

Stopping criterion: This parameter dictates when the algorithm should cease searching the solution space. We decided to stop the algorithm when there is no improvement for the last 20 iterations. This result will be explained by the help of Table 11 later.

Tabu tenure: This parameter dictates the number of iterations for which a move should remain in the tabu list after inclusion. We considered 5 levels of tabu tenure in our pilot runs that are conducted for fine tuning purposes: 1, 5, 7, 10, and 15.

We considered five objective functions in our pilot runs. Total tardiness, total flow time, and total system slack of the initial schedule; total tardiness and total flow time of the estimated realization of initial schedule by using Method 1. Note that Method 1 uses the breakdown distributions. Instead of using the breakdown parameters presented above, which would lead to four Method 1 variants, we used an average breakdown scheme to keep the pilot runs simple: we set $\theta = 6$ and $\beta = 1$. Similarly we used just the expectation instead of $\mu L + (1-\mu)U$ as busy time period, because the optimal level for μ is not determined yet (which will be done next).

We generate a single instance from each problem class, resulting in 5000 runs (200 problem combinations (See Table 9) x 5 tabu tenure levels x 5 objective

functions). We run the proposed tabu search algorithm on each problem for 1000 iterations. We deliberately did not include Method 2 in our pilot runs because the numerical integration of the density function of Gamma distribution for a lot times (needed to calculate relevant probabilities of Method 2) was too time demanding for 1000 iterations. We observed the number of iterations between best solution improvements and recorded the maximum one (*M*) for each problem. We also observed the best objective value attained for each problem. The results are presented in Table 11 and Table 12.

Table 11. The Maximum Number of Iterations between Best Solution Improvements

M	Coverage Percentage
10	87.26
20	96.26
30	97.44
40	97.96
50	98.3
60	98.48
70	98.6
80	98.76
90	98.82
100	98.96
150	99.22
200	99.48

In Table 11, *M* column lists the maximum number of iterations between best solution improvements. The coverage percentage gives the percentage of the problems that reach the optimal solution (the best solution found so far at the end of 1000 iterations) for corresponding *M* value. For example, if we stop the algorithm when there is no improvement for last 10 iterations, we still find the "so far best" solution (i.e. the best solution obtained within 1000 iterations) for the 87.26 percent of the 5000 pilot problems. Upon investigation of Table 11, we decided to use 20 iterations for the value of *M*.

Table 12. Tabu Tenure Comparisons

	Objective Type				
Tenure	**Tardiness**	**Flow Time**	**Slack***	**Tardiness (Method 1)**	**Flow time (Method 1)**
1	1873.985	2506.48	3723.615	2543.03	3147.135
5	1879.59	2481.985	3816.555	2491.02	3071.08
7	1884.035	2482.245	3963.945	2533.325	3164.69
10	1852.58	2502.47	3999.325	2456.855	3105.915
15	1857.605	2447.335	4087.06	2495.42	3076.48
	Objective Type				
Rankings	**Tardiness**	**Flow Time**	**Slack***	**Tardiness (Method 1)**	**Flow time (Method 1)**
1st	10	15	15	10	5
2nd	15	5	10	5	15
3rd	1	7	7	15	10
4th	5	10	5	1	1
5th	7	1	1	7	7

*** = Higher slack values are better**

First part of Table 12 presents the average value of the objective function used for the tabu tenure – neighborhood evaluation combination. Second part ranks the tabu tenures for each neighborhood evaluation mechanism, from the best to the worst. The best two alternatives for the tabu tenure are 10 and 15. It is difficult to choose a clear winner. Hence, we decided to use random tabu tenures that are uniformly distributed between 10 and 15 for the future tests.

Determining the best λ value: We generated five instances for each of the 800 possible parameter combination (Table 9 and 10) for determining λ value of Method 1. We have two objective functions in TS: total tardiness and total flow time of the estimated realizations. We used 0.2, 0.4, 0.6 and 0.8 as candidate λ values. We also used $E[f(t)]$ instead of. $\lambda L + (1-\lambda)U$ as the busy time period of the machine, for the sake of comparison. We simulated the "so far best" schedule for 5 times, and recorded the average objective function values as well as the average of Method 1 estimates for these values. We also conducted a simple linear regression analysis. The dependent variable is average of average objective value of 5 simulation replications of 5 problem instances that have the same problem parameters. The independent variable is the

average Method 1 estimates of objective values. The R^2 (coefficient of determination) values are also recorded. Tables 20-21 (in appendix) and Tables 22-23 (in appendix) present the results for total tardiness and total flow time, respectively.

Since coefficients of determinations are very close to 1.0 and to each other, we focus on error percentages as we assess the quality of different λ levels. The following observations are true for both total tardiness total flow time, for all levels of λ.

The percentage errors for B1 and B3 (long repair duration) are greater than the percentage errors for B2 and B4 (short repair duration). This means that the estimation error is greater in the case of major breakdowns.

As number of jobs increases, the percentage errors decrease. This is an expected result. As the number of jobs increases, we expect the number of machine failures to increase, i.e., we get more observations from the breakdown distribution, consequently variance of observations decreases and prediction quality increases (similar to the notion of Law of Large Numbers).

As arrival parameter α increases (as the range of jobs arrival times gets wider) the quality of prediction gets worse.

As the processing time variability increases, the percentage errors also increase, but the difference is not so significant.

When the system performance of interest is total tardiness, due date schemes with low tightness variation (D2 and D3) tend to increase the quality of prediction, but the difference is not too much.

As seen in the Tables 20-23, although $\lambda = 0.8$ gives the best overall prediction (both, in terms of coefficient of determination and in terms of percentage error) for both objective functions, its quality is the worst in terms of realized schedule performance. $\lambda = 0.4$ for total tardiness and $\lambda = 0.6$ for total flow time are the best in terms of realized schedule quality. The performance of $\lambda = 0.6$ for the total tardiness case is very close to that of $\lambda = 0.4$, so $\lambda = 0.6$ is chosen as the overall best value.

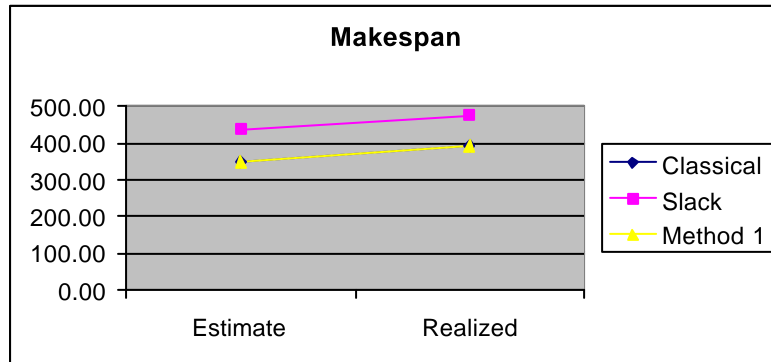
5.3. Comparison of Method 1 with Other Approaches

5.3.1. Robustness

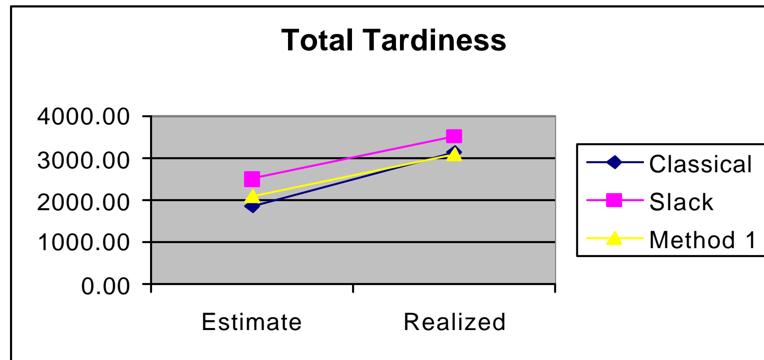
We compared the performance of Method 1, from the viewpoint of robustness, to the approach of using average system slack. We also used classical approach, which minimizes planned (initial) performance measure, as a benchmark. Leon *et al.* (1994) estimates the realized performance measure as follows:

$E[f(S)] = f(S_0) + E[\boldsymbol{d}(S)]$ *where* $E[f(S)]$ is the expectation of the realized performance measure (our robustness measure); $f(S_0)$ is the initial (or planned) performance measure of S, and $E[\boldsymbol{d}(S)]$ is the degradation in the performance measure because of the disruptions, which is estimated by using –(average system slack) as a surrogate measure.

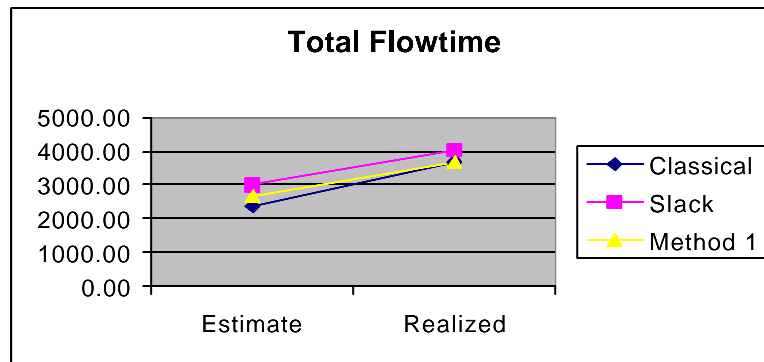
We generated five instances for each of 800 possible problem classes (see Tables 9-10), leading to 4000 experimental design points. We run our TS algorithm with 3 different neighbourhood evaluation functions (classical approach, slack method, and Method 1) for 3 performance measures (makespan, total tardiness, and total flow time). This resulted in 4000 x 3 x 3 = 36000 runs. We recorded the estimated performance measures, the realized performance measures (taken as the average of 5 Monte-Carlo simulations), and stability measures (total absolute job completion time differences). The results are presented in Tables 24, 25, and 26 (in appendix) for makespan, total tardiness, and total flow time performance measures, respectively. Figure 10 is also prepared to display estimated and realized performance measures for makespan, total tardiness, and total flow time, respectively. Table 13 presents a summary of the results. Note that the stability values are also given even though our objective is to optimize robustness. The figures marked with + sign are the worst in their group, whereas the figures marked with a * sign are the best according to the paired-t tests with $\alpha = 0.05$.



a) Makespan Estimate-Realized Graph



b) Total Tardiness Estimate-Realized Graph



c) Total Flow Time Estimate-Realized Graph

Figure 10. Estimate vs Realized Performance Measure Plots (Robustness)

Table 13. Summary Table for Robustness Results

		Makespan	Total Tardiness	Total Flow Time
Robustness	Classical Approach	391.08	3134.52	3712.21
	Slack Method	474.97+	3509.84+	4058.31+
	Method 1	390.65	3083.87*	3697.60
Stability	Classical Approach	1472.48	1316.40+	1338.23+
	Slack Method	1030.04*	1051.61	1054.15
	Method 1	1452.09	1044.33	1035.42*

After careful analysis of the results presented in these tables and Figure 10, we can make the following observations:

In terms of robustness, the average slack method is not good at all (it is statistically significantly worse than Method 1 and classical approach) for all three performance measures. The proposed method (Method 1) is better than both the average slack method and the classical approach. However, the difference between the proposed method and other two methods are significant for only the total tardiness criterion (i.e., Method 1 yields smaller numerical values than these two methods but the difference is not statistically significant for makespan and total flow time criteria).

In terms of stability the classical approach is the worst (the difference is significant for total flow time and total tardiness criteria). In general Method 1 and the average slack method are very competitive. Even though the average slack method is better than Method 1 for makespan criterion, Method 1 is better when the other two criteria are considered (total tardiness and total flow time). The improved performance of the average slack method over Method 1 can be attributed due to the following fact: as stated before, the average slack method estimates expected makespan (the robustness measure) as $E[f(S)] = f(S_0) + E[\boldsymbol{d}(S)]$. The initial makespan of *any acceptable schedule* (i.e., a schedule that does not keep machine idle when there is a job waiting in the queue) is constant. Therefore, minimizing $E[f(S)] = f(S_0) + E[\boldsymbol{d}(S)]$ is equivalent

to minimizing $E[\boldsymbol{d}(S)]$, which is estimated by –(average system slack). As a result, the average system slack method optimizes robustness by maximizing average system slack for makespan criterion, which is exactly the same approach that is used to optimize stability by this method. This fact can be confirmed by comparing the average slack method – makespan critetion cells in Table 13 with the ones in Table 14, which presents a summary when the primary goal is to optimize stability. The average slack method (hopefully) optimizes stability inherently while (hopefully) minimizing expected makespan (robustness measure), whereas Method 1 does not consider stability when it is used to optimize robustness. This is why the average slack method is better than Method 1 in terms of stability for the makespan criterion when the primary concern is robustness.

In short, we conclude that the proposed method (Method 1) is generally competitive for both robustness and stability.

5.3.2. Stability

In this section, we compare the performance of Method 1 and the average system slack method. Again, we used classical approach (which only minimizes planned or initial performance measure, and does not consider stability) as a benchmark.

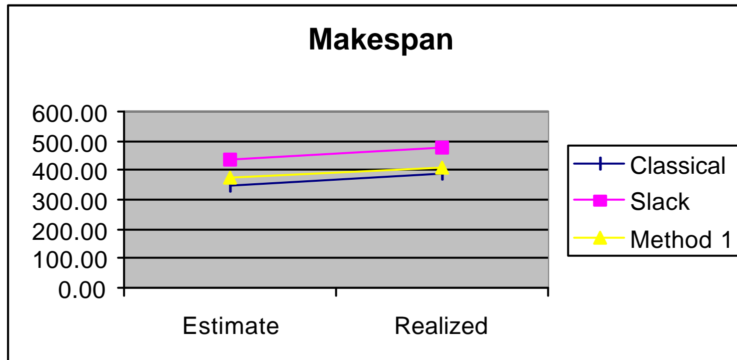
As in the case of robustness we have 800 possible problem classes resulted in 4000 design points. We run our TS algorithm for 3 different neighbourhood evaluation functions (classical approach, slack method, and Method 1). We recorded the estimated performance measures, the realized performance measures (taken as the average of 5 Monte-Carlo simulations), and stability measures (total absolute job completion time differences). The results are presented in Tables 27, 28 and 29 (in appendix) for makespan, total tardiness, and total flow time performance measures, respectively. Table 14 presents a summary of the results. The figures marked with + sign are the worst in their group, whereas the figures marked with a * sign are the best according to the paired-t tests with $\alpha = 0.05$. Figure 11 depicts estimated and realized performance

measures for makespan, total tardiness, and total flow time. We make the following observations from the results:

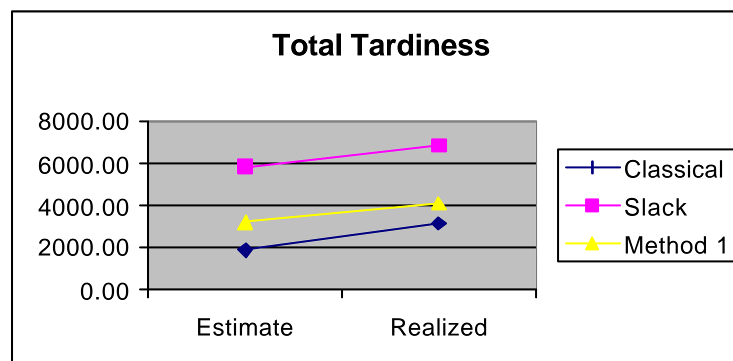
In terms of stability, which is the primary goal in these experiments, the proposed method (Method 1) is significantly better than the average slack method and the classical approach. Hence, it is the winner. When we look at the secondary goal (robustness), however, the classical approach is significantly better than the other two methods (the average slack method and Method 1). This can be explained as follows: as it is discussed in the previous section, the classical approach performs well when the primary objective is to optimize robustness. We see that the robustness performance of average slack method and Method 1 deteriorate when we change the primary goal to optimizing stability. However the classical approach does not suffer this deterioration, as it does not consider stability at all. As a result, it keeps it high robustness performance. These observations give us an evidence of the existence of a trade-off between robustness and stability. This trade-off is further discussed in section 5.3.4.

Table 14. Summary table for stability results

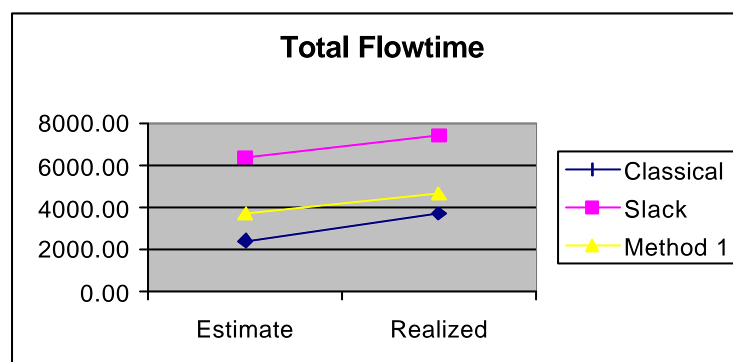
		Makespan	**Total Tardiness**	**Total Flow Time**
Robustness	**Classical Approach**	391.13^{*}	3131.31^{*}	3713.44^{*}
	Slack Method	478.05^{+}	6843.11^{+}	7400.27^{+}
	Method 1	408.36	4070.64	4614.69
Stability	**Classical Approach**	1480.48^{+}	1321.59^{+}	1320.97^{+}
	Slack Method	1040.26	1040.26	1040.26
	Method 1	916.03^{*}	916.03^{*}	916.03^{*}



a) Makespan Estimate-Realized Graph



b) Total Tardiness Estimate-Realized Graph



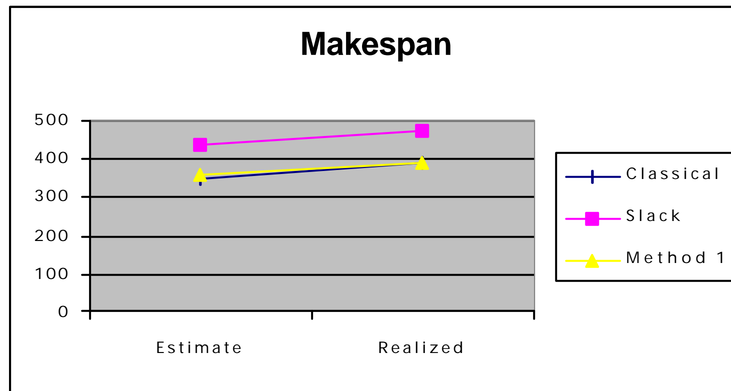
c) Total Flow Time Estimate-Realized Graph

Figure 11. Estimate vs Realized Performance Measure Plots (Stability)

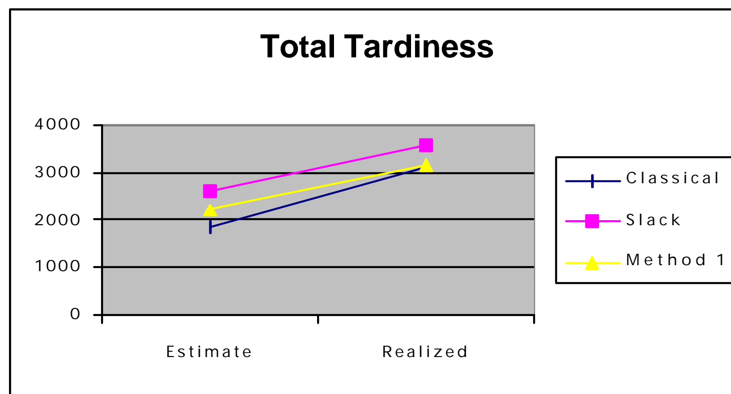
5.3.3. Bicriterion Approach

In fact, the objective function that we minimize can be viewed as w*robustness measure + (1-w)*stability measure, where w is a real number between 0 and 1 that represents the weight given to a particular measure. For section 5.3.1, w is equal to 1 (pure robustness) and for section 5.3.2, w is equal to 0 (pure stability). We set up an experimental study to check if it is possible to maintain good expected realized schedule performance (robustness) while increasing schedule stability significantly when w is close to 1. As Leon *et al.* (1994) did for a similar study, we take $w = 0.85$. Hence, we used 0.85*(Robustness measure) + 0.15*(Stability measure) as the neighbourhood evaluation function of TS algorithm for both, slack method and Method 1. We also included classical approach in our study for the benchmark purposes.

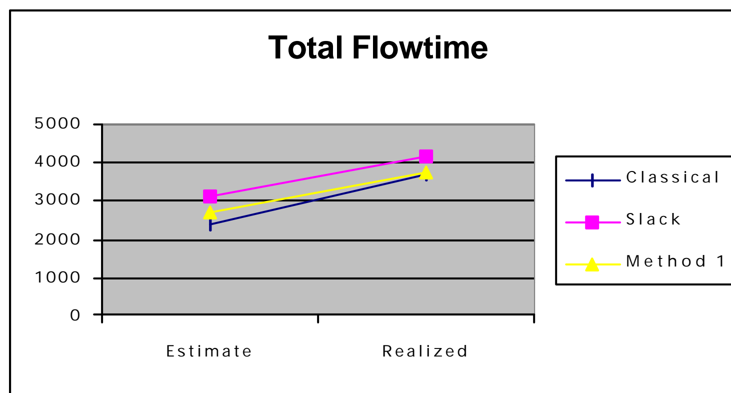
We generated five instances for each of 800 possible problem classes. We recorded the estimated performance measures, the realized performance measures (taken as the average of 5 Monte-Carlo simulations), and stability measures (total absolute job completion time differences). The results are presented in Tables 30, 31 and 32 (in appendix) for makespan, total tardiness, and total flow time performance measures, respectively. Table 15 presents a summary of the results. Table 16 gives (0.85 * robustness + 0.15 * stability) values. The figures marked with + sign are the worst in their group, whereas the figures marked with a * sign are the best according to the paired-t tests with $\alpha = 0.05$ in both tables. Figure 12 depicts estimated and realized performance measures for makespan, total tardiness, and total flow time.



a) Makespan Estimate-Realized Graph



b) Total Tardiness Estimate-Realized Graph



c) Total Flow Time Estimate-Realized Graph

Figure 12. Estimate vs Realized Performance Measure Plots (Bicriterion)

Table 15. Summary Table for Bicriterion Approach Results

		Makespan	Total Tardiness	Total Flow Time
Robustness	Classical Approach	390.76	3116.18*	3718.98
	Slack Method	475.08+	3590.70+	4144.81+
	Method 1	392.64	3169.73	3728.50
Stability	Classical Approach	1473.71+	1312.24+	1333.37+
	Slack Method	1023.92	1011.84	1012.18
	Method 1	1033.53	1015.76	1010.21

Having carefully analyzed these results, we make the following observations:

In terms of robustness, the average slack method is statistically the worst for all three performance measures. The classical approach is slightly better than the proposed method (Method 1) for all three performance measures, but the difference is statistically significant for only the total tardiness criterion (i.e., the classical approach yields smaller numerical values for the makespan and total flow time criteria, but the difference is not statistically significant).

In terms of stability, the classical approach is not good at all (it is significantly worse than the average slack method and Method 1). In general, Method 1 and the average slack method are very competitive. Even though the average slack method is better than Method 1 for the makespan and total tardiness criteria, Method 1 is better when the total flow time criterion is considered. The differences between these two methods are not statistically significant for all three performance measures.

We see that the proposed Method (Method 1) is not better than the other methods (the classical approach and the average slack method) if robustness and stability are considered separately. However, when we look at values of composite objective function (Table 16), we see that the proposed method (Method 1) performs well. It is better than other two methods (the average slack method and the classical approach) for the makespan and total flow time criteria, and the differences are statistically

significant. Although the classical approach yields slightly better a numerical value for total tardiness criteria, the difference is not significant.

Table 16. Composite Objective Function (0.85*Robustness + 0.15*Stability) Values

	Makespan	Total Tardiness	Total Flow Time
Classical Approach	553.20	2845.59	3361.14
Slack Method	557.41	3203.87^{+}	3674.92^{+}
Method 1	488.77*	2846.63	3320.76*

To summarize, we conclude that Method 1 is generally better than the average slack method and the classical approach when a bicriterion approach that considers both the stability and robustness measures is used. Further, we see that it is possible to maintain high realized performance values (robustness) while increasing stability significantly by using a bicritetion approach that gives slight weights to stability. The relationship between robustness and stability is further discussed in the next section.

5.3.4. Robustness vs Stability

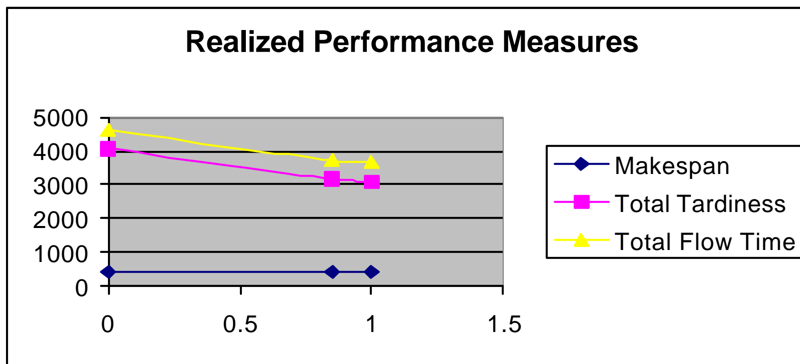
In this section, we further elaborate (or discuss) the performance of Method 1 for varying values of *r*. Our experimental results indicate that Method 1 performs generally better than the other methods the concern is to minimize (r * robustness + (1-r) * stability) measure, for r=0, r=0.85, and r=1.0 (Sections 5.3.1, 5.3.2, and 5.3.3). Tables 17-18, and Figure 13 summarize the experimental results.

Table 17. Robustness under Method 1 (Summary)

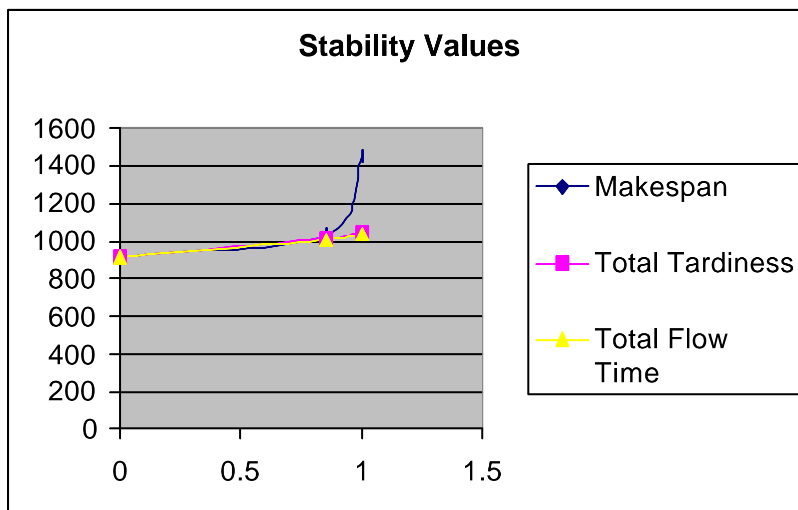
	Robustness Values		
	r=0	**r=0**	**r=0.85**
Makespan	408.36	408.36	392.64
Total Tardiness	4070.64	4070.64	3169.73
Total Flow Time	4614.69	4614.69	3728.50

Table 18. Stability Values under Method 1 (Summary)

	Stability Values		
	r=0	r=0.85	r=1.0
Makespan	916.03	1033.53	1452.09
Total Tardiness	916.03	1015.76	1044.33
Total Flow Time	916.03	1010.21	1035.42



a) Realized Performance Measures



b) Stability Values

Figure 13. Robustness vs Stability

The following observations can be made from the results:

For the makespan criterion, realized performance measure (robustness) is fairly constant, (i.e., the robustness curve is nearly horizontal). That is, as we change the value of r, we do not affect realized makespan at all. Similarly as we examine the stability curve, we see that it abruptly deteriorates at $r = 0.85$.

For the remaining two performance measures (total tardiness, and total flow time criteria) we see that the curves for both, robustness and stability are nearly linear. That is, as we gradually increase r, we gradually improve robustness, whereas stability gradually deteriorates. The absolute value of slope of robustness line ($\approx$ 990) is greater than the slope of stability line ($\approx$ 120). We see that robustness is more sensitive to the changes in r. Moreover, the lines are nearly parallel to each other; that is, the trade-off between robustness and stability is independent from performance measures.

From the above two observations, we conclude that the practitioners should emphasis on stability, rather than robustness for the makespan criterion. For the remaining two performance measures (i.e., total tardiness and total flow time) there is a linear trade-off (*e.g.,* if we increase r by Δr, we gain $990\Delta r$ units of robustness, whereas we loose $120\Delta r$ units of stability for our problem set).

5.4. Comparison of Method 2 with Other Approaches

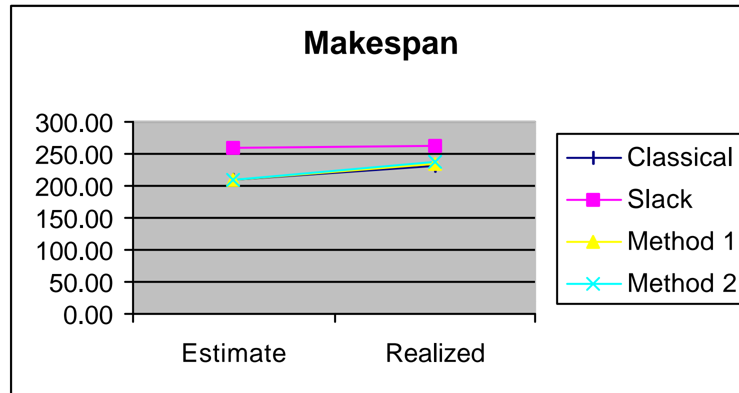
We compared the performance of Method 2, from the viewpoint of robustness, to the average system slack and Method 1. We also used classical approach, which minimizes planned (initial) performance measure, as a benchmark. As stated in Section 4.2.3.2, Method 2 assumes a single machine breakdown for the whole scheduling period. Because of this restrictive assumption, we decided to use a continuous scheduling scheme that reschedules the system from scratch after each machine breakdown (Response BR3 of Table 2).

We excluded 70-job and 90-job problems due to high computational burden of Method 2. We generated five instances for each of the remaining 480 possible problem

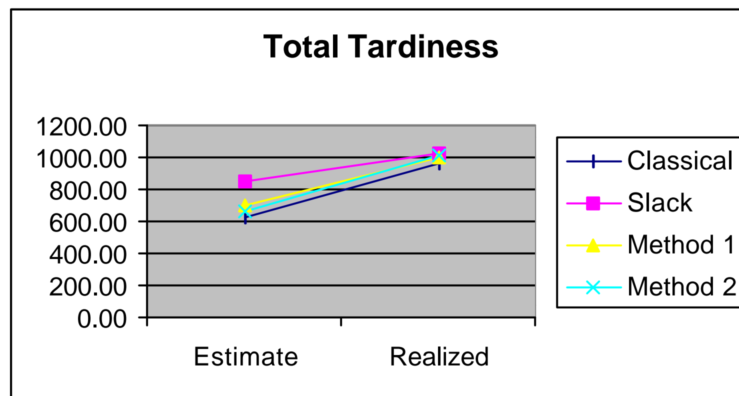
classes (see Tables 9-10), leading to 2400 experimental design points. We run our TS algorithm with 4 different neighbourhood evaluation functions (classical approach, slack method, Method 1, and Method 2) for 3 performance measures (makespan, total tardiness, and total flow time). This resulted in 2400 x 4 x 3 = 28800 runs. After a machine breakdown remaining jobs are rescheduled from scratch. We recorded the estimated performance measures, the realized performance measures, and stability measures (total absolute job completion time differences). The results are presented in Tables 33, 34, and 35 (in appendix) for makespan, total tardiness, and total flow time performance measures, respectively. Figure 14 is also prepared to display estimated and realized performance measures for makespan, total tardiness, and total flow time, respectively. Table 19 presents a summary of the results. Note that the stability values are also given even though our objective is to optimize robustness. We compared the best two methods with each other for each group of Table 18. If the difference is statistically significant according to paired-t test with $\alpha = 0.05$, the figure that summarizes the performance of that method is marked with a * sign. Similarly, if the difference between the wost two methods is statistically significant, we marked the corresponding figure with a + sign.

Table 19. Summary Table for Continuous Scheduling Results (Robustness)

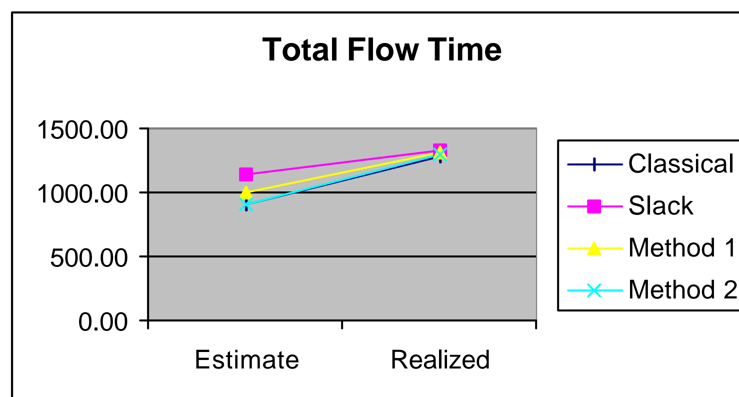
		Makespan	**Total Tardiness**	**Total Flow Time**
Robustness	**Classical Approach**	232.81	961.24*	1278.14
	Slack Method	262.86+	1026.97	1328.53
	Method 1	234.21	1000.80	1317.87
	Method 2	237.61	1012.00	1296.14
Stability	**Classical Approach**	487.06	414.10	455.25
	Slack Method	912.15	479.40+	481.08
	Method 1	494.85	401.06	437.12
	Method 2	1096.83+	426.03	439.19



a) Makespan Estime-Realized Plot (Continuos Scheduling)



b) Total Tardiness Estimate-Realized Plot (Continuous Scheduling)



c) Total Flow Time Estimate-Realized Graph (Continuous Scheduling)

Figure 14. Estimate vs Realized Performance Measure Plots (Continuous Scheduling)

After careful analysis of the results presented in these tables and Figure 14, we can make the following observations:

In terms of robustness, the average slack method is not good at all for all three performance measures. It yields the worst numerical results, but this is statistically significant for only the makespan criterion. The classical approach yields the best numerical values for all the three performance measures, and it is statistically the best for the total tardiness criterion. The performance of the proposed methods, Method 1 and Method 2, lies in between. Method 1 is better than Method 2 when the makespan and total tardiness criteria are considered, whereas Method 2 is better than Method 1 for the total flow time criteria. Nonetheless, the difference between Method 1 and Method 2 is statistically significant for only the makespan criterion. The performance degradation of Method 1 and inefficiency of Method 2 can be attributed to the fact that both methods inherently assume right shift response to machine breakdowns, whereas in reality after each machine breakdown the system is rescheduled from the scratch.

In terms of stability, Method 2 and the average slack method do not perform well. Method 2 is statistically the worst for the makespan criterion, and the average slack method is statistically worst for the total tardiness criterion. The average slack method is also numerically the worst for total flow time criterion, but this is not statistically significant. Method 1 and the classical approach seem to perform well, and these two methods are competitive. The classical approach is better for the makespan criterion whereas Method 1 yields better numerical results for the remaining two performance measures (total tardiness and total flow time). Nonetheless, none of these differences is statistically significant.

In short, we conclude that the classical approach is better and the both proposed methods are inefficient for continuous scheduling scheme. Thus, better surrogate measures should be developed for continuous scheduling environment. Note that, as Church and Uzsoy (1992) discuss, the benefit of extra scheduling diminish rapidly. Moreover scheduling after each machine breakdown creates system nervousness. We believe that continuous scheduling under an adaptive scheduling approach (where scheduling process is triggered after a certain amount of deviation from the initial

schedule) the proposed methods, especially Method 1, can maintain its high performance. This topic requires investigation, and can be viewed as a further research direction.

Chapter 6

Conclusion

In this thesis, we first addressed the notions of robustness and stability. We defined the reactive scheduling problem with our notations and terms. Then, we defined some measures for robustness (and stability) and pointed out some possible policies. After that, an extensive literature review takes place. On seeing the exact measures are difficult to calculate, we developed two new surrogate measures. Finally, we embedded these surrogate measures into a TS algorithm to generate robust and stable schedules for a single machine subject to random machine breakdowns.

We compared Method 1 with the other approaches (the average slack method and the classical approach) using a periodic scheduling where type of response to machine breakdowns is to right shift the remaining jobs. We first compared the alternative approaches for a viewpoint of robustness, taking stability as a secondary goal. Our experimental results show that the average slack method is not good at all, and Method 1 is generally better than the classical approach. In addition, it yields significantly more stable schedules than the average slack method for the total tardiness and total flow time criteria.

Next, we compared the alternatives approach from a viewpoint of stability, taking robustness as a secondary objective. We see that Method 1 generates more stable schedules than its alternatives for all three performance measures. We also observed that the classical approach does not perform well in terms of stability (statistically the worst alternative), however it is statistically the best alternative in terms of robustness.

Then, we wondered if it is possible to maintain high robustness and improve stability significantly when we use a bicriterion approach that considers both robustness and stability. We used 0.85 * robustness + 0.15 * stability as the neighbourhood evaluation function in TS algorithm. We observed that the classical approach is generally better in terms of robustness and the average slack method is generally better in terms of stability although in both robustness and stability Method 1 is competitive. However, we observe that when we evaluate the composite objection function, which considers both robustness and stability simultaneously, Method 1 is generally better. In addition we found out that it is possible to maintain high robustness while improving stability significantly by means of this bicriterion approach, even when the weight of the stability is as small as 0.15.

Next, we further analyze the performance of Method 1 with varying weights (r) given to robustness. We observed that for the makespan criterion, the practitioners should emphasize on stability because robustness is insensitive to the values of r. We also observed that there is a linear trade-off between robustness and stability (i.e., improving one deteriorates the other). Our experimental results show that stability is less sensitive to the changes in r than robustness (i.e, the absolute value of the slope of the line that plots stability values against varying values of r is less than that of robustness).

Finally, we compared the performance of Method 2 with the other alternatives (the classical approach, the average slack method and Method 1) from the viewpoint of robustness, taking stability as the secondary goal. Since Method 2 assumes a single machine breakdown through the whole scheduling period, we compared the alternatives in a continuos scheduling environment where the whole system is rescheduled from the scratch after a machine breakdown. Our computational experiments indicate that the performance of Method 1 deteriorates in a continuous scheduling environment, and Method 2 is rather inefficient. In terms of robustness the classical approach yields the best numerical values and this fact is statistically significant for the total tardiness criterion. In terms of stability, which is the secondary goal, Method 1 and the classical

approach are competitive. The classical approach is better for the makespan criterion, whereas Method 1 is better for the total tardiness and total flow time criteria.

We completed this study under a static environment with nonzero jobs arrival times (can be called as semi-dynamic). We do not know the performance of our proposed methods in a true dynamic environment. We also do not know the stability performance of the proposed methods in a continuous scheduling environment, and better surrogate measures for robustness should be developed for the same environment. These three topics can be viewed as further research directions from where we stand.

Bibliography

[1] Adams, J., Balas, E., and Zawack, D. "The Shifting Bottleneck Procedure for Job Shop Scheduling," *Management Science*, 34 (1988), 391-401

[2] Akturk, M. S., and Gorgulu, E., "Match-up Scheduling under a Machine Breakdown," *European Journal of Operational Research*, 112 (1999), 81-97

[3] Aytug, H., Lawley, M., McKay, K., Mohan, S., and Uzsoy, R. "Executing Production Schedules in the Face of Uncertainties: A Review and Some Future Directions," Technical Report, Purdue University, 2001.

[4] Church, L. K., and Uzsoy, R., "Analysis of Periodic and Event-Driven Rescheduling Policies in Dynamic Shops," *International Journal of Computer Integrated Manufacturing*, 5 (1992), 153-163

[5] Applegate, D., and Cook, W., "A Computational Study of Job Shop Scheduling," Technical Report, CMU-CS-90-145, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, (1990).

[6] Carlier, J., "The One-Machine Sequencing Problem," *European Journal of Operational Research*, 11 (1982), 42-47

[7] Daniels, R. L, and Kouvelis, P., "Robust Scheduling to Hedge Against Processing Time Uncertainty in Single-Stage Production," *Management Science*, 41.2 (1995), 363-376

[8] Glover, F. W., and Laguna M., *Tabu Search.*, Kluwer Academic Publishers, Boston, 1997.

[9] Kempf, K., Uzsoy, R., Smith, S., and Gary, K. "Evaluation and Comparison of Production Schedules," *Computers in Industry*, 42 (2000), 203-220

[10] Kutanoglu, E., and Sabuncuoglu, I. "Experimental Investigation of Iterative Simulation-Based Scheduling in a Dynamic and Stochastic Job Shop," *Journal of Manufacturing Systems,* 20.4 (2001), 264-279

[11] Laguna, M., Barnes, J. W., and Glover, F. W. "Tabu Search Methods for a Single Machine Scheduling Problem," *Journal of Intelligent Manufacturing*, 2 (1991), 63-74

[12] Law, A. M., Kelton, W. D., *Simulation Modeling and Analysis*. 2nd ed., McGraw-Hill, Inc., Singapore, 1991.

[13] Lawrance, S., "Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques," GSIA, Carnegie Mellon University, (1984).

[14] Leon, V. J., Wu, S. D., and Storer, R. H., "Robustness Measures and Robust Scheduling for Job Shops," *IIE Transactions*, 26.5 (1994), 32-43

[15] Mehta, S. V., and Uzsoy, R., "Predictable Scheduling of a Job Shop Subject to Breakdowns," *IEEE Transactions on Robotics and Automation*, 14.3 (1998), 365-378

[16] Mehta, S. V., and Uzsoy, R., "Predictable Scheduling of a Single Machine Subject to Breakdowns," *Int. J. Computer Integrated Manufacturing*, 12.1 (1999), 15-38

[17] O'Donovan, R., Uzsoy, R., and McKay, K. N., "Predictable Scheduling of a Single Machine with Breakdowns and Sensitive Jobs," *Int. J. Prod. Res.*, 37.18 (1999), 4217-4233

[18] Pinedo, M., *Scheduling: Theory, Algorithms, and Systems.* Prentice-Hall, Inc., New Jersey, 1995.

[19] Rachamadugu, R. V., and Morton, T. E., "Myopic Heuristics for the Single Machine Weighted Tardiness Problem," Working Paper 30-82-83, Graduate School of Industrial Administration, Carnegie Mellon University, (1982).

[20] Sabuncuoglu, I., and Bayiz, M., "Analysis of Reactive Scheduling Problems in a Job Shop Environment," *European Journal of Operational Research*, 126 (2000), 567-586

[21] Sabuncuoglu, I., and Hommertzheim, D. L., "Dynamic Dispatching Algorithm for Scheduling Machines and AGVs in a Flexible Manufacturing System," *International Journal of Production Research*, 30 (1992), 1059-1080

[22] Sabuncuoglu, I., and Karabuk, S., "Rescheduling Frequency in an FMS with Uncertain Processing Times and Unreliable Machines," *Journal of Manufacturing Systems*, 18.4 (1999), 1-16

[23] Wu, S. D., Byeon, E., and Storer, R. H., "A Graph-Theoretic Decomposition of the Job Shop Scheduling Problem to Achieve Scheduling Robustness," *Operations Research*, 47.1 (1999), 113-124

[24] Wu, S. D., Storer, R. N., and Chang P., “One-Machine Rescheduling Heuristics with Efficiency and Stability as Criteria,” *Computers Ops Res.*, 20.1 (1993), 1-14

[25] Yamamoto, M., and Nof, S. Y., “Scheduling/Rescheduling in a Manufacturing Operating System Environment,” *International Journal of Production Research*, 23.4 (1985), 705-722

Appendix

Table 20. Comparison of λ Values - Total Tardiness Case

Parameter	Use mean		Use $\lambda L + (1-\lambda)U$							
			$\lambda = 0.2$		$\lambda = 0.4$		$\lambda = 0.6$		$\lambda = 0.8$	
	Sim. Result	Estimate	Sim. Result	Estimate	Sim. Result	Estimate	Sim. Result	Estimate	Sim. Result	Estimate
Breakdown										
B1	2754.77	2428.57	2694.11	1982.04	2691.25	2039.42	2693.71	2162.13	2780.47	2541.50
B2	2080.04	2019.88	2075.30	1908.07	2074.96	1920.43	2078.28	1953.15	2087.51	2044.91
B3	4905.10	4160.70	3178.05	2485.31	4727.22	2649.24	4716.50	3103.20	5008.04	4740.85
B4	2536.08	2412.52	2519.90	2012.87	2507.06	2072.47	2516.25	2179.85	2555.77	2529.09
Number of Jobs										
10	89.85	62.06	90.24	45.32	88.48	48.35	88.29	52.30	89.13	70.16
30	826.38	701.64	804.86	512.53	813.36	539.49	806.54	588.80	828.65	757.60
50	2316.70	2055.74	2290.58	1541.79	2247.56	1602.54	2272.87	1739.38	2354.70	2212.00
70	4550.99	4076.67	4442.40	3084.39	4457.42	3213.88	4444.71	3482.90	4617.67	4397.59
90	7561.07	6880.99	7455.63	5230.48	7393.79	5447.68	7393.52	5884.54	7649.58	7383.08

Table 20. (Cont'd)

Arrival parameter a										
0.25	6528.37	6214.33	6481.94	5194.58	6447.14	5332.71	6470.32	5609.63	6540.48	6491.04
0.50	4695.69	4310.33	4610.82	3314.72	4619.36	3452.23	4592.97	3729.03	4747.29	4596.94
0.75	2946.52	2499.01	2848.66	1560.35	2801.48	1688.24	2810.39	1945.30	3006.75	2782.52
1.25	837.63	545.60	811.49	223.74	801.14	249.13	807.86	314.98	881.51	698.37
1.75	336.77	207.82	330.79	121.13	331.48	129.64	324.38	148.97	363.71	251.56
Process Time										
P1	2772.61	2449.06	2759.46	1854.72	2734.15	1931.83	2720.15	2085.05	2807.32	2624.58
P2	3365.39	3061.78	3274.02	2311.08	3266.09	2408.95	3282.22	2614.11	3408.57	3303.59
Due Date										
D1	3393.83	3043.31	3337.69	2361.68	3314.88	2449.86	3298.89	2630.18	3433.99	3262.93
D2	3370.61	3047.28	3315.24	2349.08	3317.26	2442.18	3290.63	2627.67	3437.72	3256.15
D3	2663.79	2389.93	2616.12	1751.74	2595.56	1835.61	2626.24	2003.67	2677.80	2585.48
D4	2847.76	2541.16	2797.91	1869.11	2772.77	1953.91	2788.98	2136.81	2882.27	2751.78
Overall	3069.00	2755.42	3016.74	2082.90	3000.12	2170.39	3001.18	2349.58	3107.95	2964.09

Table 21. Comparison of λ Values - Total Tardiness Case (Error Percentage - R^2 values)

Parameter	Use mean		Use $\lambda L + (1-\lambda)U$							
			$\lambda = 0.2$		$\lambda = 0.4$		$\lambda = 0.6$		$\lambda = 0.8$	
	Error Perc.	Coeff. Det.	Error Perc.	Coeff. Det.	Error Perc.	Coeff. Det.	Error Perc.	Coeff. Det.	Error Perc.	Coeff. Det.
Breakdown										
B1	11.84	0.9969	26.43	0.9868	24.22	0.9888	19.73	0.9929	8.59	0.9969
B2	2.89	0.9997	8.06	0.9989	7.45	0.9992	6.02	0.9992	2.04	0.9997
B3	15.18	0.9926	21.80	0.9472	43.96	0.9611	34.21	0.9753	5.34	0.9959
B4	4.87	0.9991	20.12	0.9927	17.33	0.9950	13.37	0.9974	1.04	0.9993
Number of Jobs										
10	30.93	0.9093	49.78	0.6545	45.35	0.7489	40.76	0.8209	21.28	0.9333
30	15.10	0.9714	36.32	0.8423	33.67	0.8639	27.00	0.9190	8.57	0.9845
50	11.26	0.9852	32.69	0.8707	28.70	0.9048	23.47	0.9433	6.06	0.9915
70	10.42	0.9872	30.57	0.8813	27.90	0.9106	21.64	0.9497	4.77	0.9944
90	8.99	0.9912	29.85	0.8856	26.32	0.9154	20.41	0.9554	3.48	0.9959

Table 21. (Cont'd)

Arrival parameter a										
0.25	4.81	0.9980	19.86	0.9527	17.29	0.9660	13.30	0.9843	0.76	0.9984
0.50	8.21	0.9961	28.11	0.9255	25.27	0.9458	18.81	0.9750	3.17	0.9972
0.75	15.19	0.9923	45.23	0.8555	39.74	0.9095	30.78	0.9616	7.46	0.9955
1.25	34.86	0.9733	72.43	0.4760	68.90	0.5796	61.01	0.7670	20.78	0.9904
1.75	38.29	0.9044	63.38	0.4984	60.89	0.5690	54.08	0.7130	30.84	0.9398
Process Time										
P1	11.67	0.9914	32.79	0.9102	29.34	0.9325	23.35	0.9633	6.51	0.9964
P2	9.02	0.9943	29.41	0.9281	26.24	0.9472	20.36	0.9717	3.08	0.9970
Due Date										
D1	10.33	0.9933	29.24	0.9261	26.10	0.9404	20.27	0.9711	4.98	0.9968
D2	9.59	0.9938	29.14	0.9204	26.38	0.9414	20.15	0.9696	5.28	0.9960
D3	10.28	0.9931	33.04	0.9189	29.28	0.9429	23.71	0.9667	3.45	0.9972
D4	10.77	0.9918	33.20	0.9164	29.53	0.9401	23.38	0.9653	4.53	0.9966
Overall	10.22	0.9930	30.96	0.9208	27.66	0.9414	21.71	0.9683	4.63	0.9965

Table 22. Comparison of λ Values - Total Tardiness Case

Parameter	Use mean		Use $\lambda L + (1-\lambda)U$							
			$\lambda = 0.2$		$\lambda = 0.4$		$\lambda = 0.6$		$\lambda = 0.8$	
	Sim. Result	Estimate	Sim. Result	Estimate	Sim. Result	Estimate	Sim. Result	Estimate	Sim. Result	Estimate
Breakdown										
B1	3373.30	3029.25	3322.13	2571.79	3344.54	2642.70	3347.93	2762.81	3414.67	3147.96
B2	2700.01	2624.63	2687.07	2503.02	2683.70	2519.19	2689.58	2552.84	2705.14	2651.24
B3	5526.36	4803.55	5470.46	3027.90	5458.64	3260.96	5370.67	3730.22	5720.83	5416.69
B4	3165.22	3014.87	3172.44	2627.48	3163.72	2681.59	3146.64	2783.35	3210.79	3150.90
Number of Jobs										
10	182.26	150.82	175.76	129.01	177.90	133.70	181.87	138.57	178.30	160.80
30	1154.50	1027.54	1137.87	820.28	1138.99	849.21	1142.19	903.74	1167.51	1088.34
50	2922.86	2638.73	2889.59	2096.57	2874.37	2159.15	2881.75	2309.28	2975.44	2807.71
70	5390.25	4917.82	5338.47	3903.24	5342.82	4037.22	5302.38	4308.36	5484.88	5240.96
90	8806.23	8105.46	8773.44	6463.64	8779.17	6701.27	8685.34	7126.58	9008.15	8660.67

Table 22. (Cont'd)

Arrival parameter a										
0.25	7248.90	6933.22	7243.61	5965.50	7245.41	6102.11	7226.83	6367.07	7313.27	7243.09
0.50	5482.04	5100.10	5498.10	4143.72	5471.94	4277.28	5449.92	4536.03	5570.92	5384.57
0.75	3658.79	3207.38	3575.67	2241.61	3595.29	2382.00	3539.80	2644.01	3770.24	3513.89
1.25	1325.81	1016.43	1290.82	614.31	1282.97	655.27	1263.64	742.97	1405.01	1172.74
1.75	740.56	583.25	706.93	447.59	717.64	463.90	713.33	496.45	754.85	644.19
Process Time										
P1	3387.03	3064.55	3411.19	2467.22	3395.88	2549.17	3356.95	2702.71	3443.60	3256.75
P2	3995.41	3671.60	3914.86	2897.87	3929.42	3003.05	3920.46	3211.90	4082.11	3926.64
Due Date										
D1	3712.63	3380.58	3677.55	2688.53	3680.33	2784.81	3640.66	2969.84	3772.09	3609.73
D2	3682.74	3371.58	3628.55	2694.32	3679.55	2788.76	3639.65	2965.65	3761.25	3592.53
D3	3677.55	3353.10	3680.67	2677.23	3639.81	2767.80	3647.99	2954.18	3748.58	3575.33
D4	3691.95	3367.03	3665.33	2670.11	3650.91	2763.07	3626.52	2939.56	3769.50	3589.18
Overall	3691.22	3368.07	3663.03	2682.55	3662.65	2776.11	3638.70	2957.31	3762.86	3591.70

Table 23. Comparison of λ Values - Total Flow Time Case (Error Percentage - R^2 values)

Parameter	Use mean		Use $\lambda L + (1-\lambda)U$							
			λ = 0.2		λ = 0.4		λ = 0.6		λ = 0.8	
	Error Perc.	Coeff. Det.	Error Perc.	Coeff. Det.	Error Perc.	Coeff. Det.	Error Perc.	Coeff. Det.	Error Perc.	Coeff. Det.
Breakdown										
B1	10.20	0.9967	22.59	0.9903	20.98	0.9911	17.48	0.9943	7.81	0.9980
B2	2.79	0.9998	6.85	0.9991	6.13	0.9994	5.08	0.9996	1.99	0.9998
B3	13.08	0.9942	44.65	0.9558	40.26	0.9635	30.54	0.9824	5.32	0.9953
B4	4.75	0.9995	17.18	0.9950	15.24	0.9958	11.55	0.9978	1.87	0.9995
Number of Jobs										
10,00	17.25	0.9169	26.60	0.6732	24.84	0.7711	23.81	0.8111	9.82	0.9471
30,00	11.00	0.9787	27.91	0.8441	25.44	0.8774	20.88	0.9324	6.78	0.9897
50,00	9.72	0.9851	27.44	0.8672	24.88	0.9004	19.87	0.9402	5.64	0.9918
70,00	8.76	0.9896	26.88	0.8824	24.44	0.9101	18.75	0.9544	4.45	0.9942
90,00	7.96	0.9929	26.33	0.8945	23.67	0.9158	17.95	0.9615	3.86	0.9958

Table 23. (Cont'd)

Arrival parameter a										
0,25	4.35	0.9984	17.64	0.9619	15.78	0.9714	11.90	0.9876	0.96	0.9982
0,50	6.97	0.9979	24.63	0.9273	21.83	0.9531	16.77	0.9768	3.35	0.9983
0,75	12.34	0.9939	37.31	0.8886	33.75	0.9193	25.31	0.9715	6.80	0.9971
1,25	23.34	0.9846	52.41	0.6296	48.93	0.6961	41.20	0.8708	16.53	0.9913
1,75	21.24	0.9313	36.69	0.6972	35.36	0.7391	30.40	0.8275	14.66	0.9764
Process Time										
P1	9.52	0.9937	27.67	0.9247	24.93	0.9438	19.49	0.9716	5.43	0.9967
P2	8.10	0.9953	25.98	0.9362	23.58	0.9482	18.07	0.9762	3.81	0.9968
Due Date										
D1	8.94	0.9947	26.89	0.9314	24.33	0.9435	18.43	0.9733	4.30	0.9971
D2	8.45	0.9949	25.75	0.9405	24.21	0.9507	18.52	0.9771	4.49	0.9966
D3	8.82	0.9942	27.26	0.9268	23.96	0.9469	19.02	0.9736	4.62	0.9968
D4	8.80	0.9946	27.15	0.9281	24.32	0.9448	18.94	0.9734	4.78	0.9963
Overall	8.75	0.9946	26.77	0.9315	24.20	0.9465	18.73	0.9744	4.55	0.9967

Table 24. Makespan Results (Robustness)

Problem Parameters	Classical			Slack			Method 1		
	Estimate	Realized	Stability	Estimate	Realized	Stability	Estimate	Realized	Stability
Breakdown									
B1	348.08	376.88	997.01	437.29	464.07	720.83	347.87	376.55	1010.35
B2	348.48	357.75	315.49	438.32	447.10	228.11	347.49	356.64	321.19
B3	348.77	451.00	3519.56	436.84	523.79	2418.46	350.96	450.84	3454.10
B4	348.31	378.68	1057.86	436.20	464.92	752.74	348.82	378.59	1022.73
Number of Jobs									
10	71.51	83.75	70.18	84.06	92.29	45.93	71.27	82.73	70.38
30	210.21	237.76	463.81	260.23	283.55	304.61	211.88	238.60	458.25
50	348.93	391.48	1174.14	436.33	473.72	799.23	348.53	390.67	1153.48
70	486.78	545.08	2187.82	614.18	666.77	1512.57	487.14	544.59	2151.46
90	624.63	697.31	3466.46	791.02	858.52	2487.85	625.11	696.67	3426.88
Arrival parameter a									
0.25	275.94	336.35	2061.56	317.43	371.42	1541.05	276.65	337.00	2016.97
0.5	277.94	336.79	2023.79	358.84	404.80	1284.53	279.18	338.52	2001.91
0.75	284.87	345.71	2013.91	398.04	436.60	1101.65	285.43	344.81	2019.24
1.25	380.10	404.57	908.32	494.88	523.60	751.01	379.44	402.08	859.24
1.75	523.20	531.96	354.83	616.62	638.43	471.95	523.22	530.87	363.09
Process time									
P1	348.65	392.17	1490.34	442.37	481.35	1037.61	349.38	391.42	1454.56
P2	348.17	389.98	1454.62	431.95	468.59	1022.46	348.19	389.89	1449.62
Due Date									
D1	349.71	392.86	1489.41	437.27	475.33	1032.37	348.59	391.97	1488.87
D2	348.05	390.46	1457.62	436.58	474.41	1024.80	348.51	389.81	1433.23
D3	347.82	389.84	1456.78	437.27	474.98	1030.56	348.68	389.64	1445.43
D4	348.05	391.15	1486.11	437.52	475.15	1032.42	349.36	391.19	1440.83
Overall	348.41	391.08	1472.48	437.16	474.97	1030.04	348.78	390.65	1452.09

Table 25. Total Tardiness Results (Robustness)

Problem Parameters	Classical			Slack			Method 1		
	Estimate	Realized	Stability	Estimate	Realized	Stability	Estimate	Realized	Stability
Breakdown									
B1	1852.86	2735.19	915.20	2491.19	3179.16	722.04	1976.09	2731.02	796.81
B2	1880.56	2144.13	278.98	2533.15	2722.89	204.76	1881.12	2114.06	251.54
B3	1839.23	4896.51	3134.87	2495.26	5013.85	2592.56	2456.47	4767.02	2407.38
B4	1865.37	2762.25	936.56	2473.72	3123.47	687.08	2056.34	2723.38	721.59
Number of Jobs									
10	47.41	100.46	62.20	58.30	106.65	56.51	47.00	95.16	58.03
30	467.59	838.15	394.43	642.03	970.46	351.63	503.43	827.93	353.75
50	1364.56	2374.75	1051.65	1834.84	2643.94	849.39	1512.09	2303.45	843.70
70	2773.92	4663.76	1947.77	3721.26	5194.05	1528.52	3116.59	4600.80	1559.50
90	4644.04	7695.48	3125.96	6235.21	8634.11	2472.00	5283.42	7592.00	2406.68
Arrival parameter a									
0.25	4790.96	6535.08	1751.66	4681.17	6398.17	1725.28	4943.69	6477.75	1549.05
0.5	2937.00	4765.64	1843.90	3396.94	4756.40	1382.29	3356.02	4721.61	1400.71
0.75	1297.56	3125.23	1858.75	2532.80	3663.86	1169.85	1777.24	2981.14	1262.38
1.25	174.34	890.11	800.87	1064.49	1633.42	639.43	270.15	885.63	701.42
1.75	97.66	356.55	326.82	816.25	1097.36	341.20	115.43	353.22	308.09
Process time									
P1	1655.13	2888.61	1276.38	2319.76	3232.11	955.16	1878.20	2798.19	968.98
P2	2063.88	3380.43	1356.42	2676.90	3787.57	1148.06	2306.80	3369.55	1119.68
Due Date									
D1	2124.73	3427.29	1316.65	2755.81	3801.33	1059.40	2399.94	3385.17	1001.19
D2	2113.25	3450.47	1344.82	2724.93	3753.44	1036.09	2354.81	3368.96	1023.41
D3	1558.56	2788.74	1312.73	2205.60	3170.32	1044.54	1734.50	2711.28	1088.71
D4	1641.47	2871.59	1291.40	2306.97	3314.28	1066.43	1880.76	2870.07	1064.01
Overall	1859.50	3134.52	1316.40	2498.33	3509.84	1051.61	2092.50	3083.87	1044.33

Table 26. Total Flow Time Results (Robustness)

Problem Parameters	Classical			Slack			Method 1		
	Estimate	Realized	Stability	Estimate	Realized	Stability	Estimate	Realized	Stability
Breakdown									
B1	2348.81	3273.53	924.72	2980.60	3666.37	685.77	2548.43	3335.51	787.08
B2	2382.96	2672.80	289.84	3021.90	3224.09	202.19	2446.21	2683.32	237.12
B3	2377.67	5581.48	3203.81	3017.62	5635.94	2618.32	3056.60	5490.58	2433.99
B4	2386.46	3321.03	934.57	2996.54	3706.85	710.31	2597.47	3280.97	683.50
Number of Jobs									
10	129.22	178.23	49.01	142.31	191.02	48.71	132.03	186.97	54.94
30	748.10	1150.60	402.49	917.16	1260.37	343.21	793.22	1141.24	348.02
50	1853.53	2897.08	1043.55	2343.23	3191.87	848.63	2079.92	2935.13	855.21
70	3471.82	5459.03	1987.22	4405.80	5956.88	1551.08	3875.29	5412.71	1537.42
90	5667.20	8876.11	3208.90	7212.32	9691.44	2479.12	6430.42	8811.93	2381.51
Arrival parameter a									
0.25	5450.40	7241.08	1790.68	5355.62	7110.10	1754.49	5693.22	7233.02	1539.80
0.5	3617.07	5472.06	1854.99	3998.73	5415.22	1416.49	4079.39	5439.05	1359.66
0.75	1843.78	3690.71	1846.93	3034.99	4158.08	1123.09	2426.59	3667.22	1240.62
1.25	551.75	1423.98	872.22	1476.80	2118.98	642.18	671.73	1373.91	702.18
1.75	406.87	733.22	326.35	1154.69	1489.19	334.50	439.94	774.78	334.83
Process time									
P1	2168.23	3463.04	1294.81	2832.85	3784.43	951.58	2429.00	3403.32	974.33
P2	2579.72	3961.38	1381.66	3175.48	4332.20	1156.72	2895.35	3991.87	1096.52
Due Date									
D1	2382.29	3720.27	1337.98	2982.84	4030.47	1047.63	2669.22	3703.06	1033.83
D2	2376.84	3718.89	1342.05	3057.06	4103.01	1045.95	2672.12	3736.74	1064.62
D3	2379.55	3698.55	1318.99	2971.35	4025.82	1054.47	2650.98	3644.90	993.91
D4	2357.22	3711.14	1353.92	3005.42	4073.97	1068.55	2656.38	3705.69	1049.31
Overall	2373.97	3712.21	1338.23	3004.17	4058.31	1054.15	2662.18	3697.60	1035.42

Table 27. Makespan Results (Stability)

Problem Parameters	Classical			Slack			Method 1		
	Estimate	Realized	Stability	Estimate	Realized	Stability	Estimate	Realized	Stability
Breakdown									
B1	349.31	377.59	1003.31	438.87	465.58	714.96	378.75	401.67	678.74
B2	348.38	357.55	323.33	440.57	449.38	232.02	374.90	381.84	196.21
B3	348.44	450.90	3558.47	438.53	528.90	2451.96	379.86	461.02	2246.87
B4	348.15	378.47	1036.82	438.86	468.36	762.09	367.84	388.89	542.31
Number of Jobs									
10	71.44	83.37	71.85	86.74	98.01	58.16	87.92	98.06	57.20
30	211.01	237.70	465.91	263.48	287.35	303.68	230.43	253.14	330.65
50	348.66	391.23	1167.71	438.93	478.15	822.49	375.60	409.18	746.29
70	485.88	543.85	2171.46	613.43	665.44	1516.27	523.43	567.22	1333.15
90	625.86	699.48	3525.50	793.46	861.32	2500.70	659.32	714.19	2112.86
Arrival parameter a									
0.25	276.68	338.18	2129.52	320.37	375.70	1575.86	306.01	360.89	1465.76
0.5	279.11	340.19	2076.84	361.13	408.31	1298.94	314.14	360.41	1191.66
0.75	283.80	342.47	1981.11	398.99	439.20	1107.18	329.36	366.54	986.35
1.25	379.41	402.38	860.22	497.38	525.98	741.15	394.92	413.60	630.34
1.75	523.85	532.42	354.72	618.17	641.08	478.16	532.25	540.34	306.05
Process time									
P1	349.04	391.54	1476.88	444.90	484.84	1045.98	379.05	411.83	854.13
P2	348.10	390.71	1484.09	433.52	471.27	1034.54	371.63	404.89	977.93
Due Date									
D1	348.59	391.00	1486.24	439.83	478.67	1037.12	374.85	408.49	929.94
D2	349.43	392.44	1499.72	440.51	479.56	1047.61	375.65	408.53	924.37
D3	348.64	391.33	1493.01	437.57	475.90	1025.38	376.01	408.77	898.55
D4	347.61	389.74	1442.96	438.93	478.08	1050.91	374.85	407.64	911.27
Overall	348.57	391.13	1480.48	439.21	478.05	1040.26	375.34	408.36	916.03

Table 28. Total Tardiness Results (Stability)

Problem Parameters	Classical			Slack			Method 1		
	Estimate	Realized	Stability	Estimate	Realized	Stability	Estimate	Realized	Stability
Breakdown									
B1	1858.57	2727.02	903.24	5824.41	6515.73	714.96	3274.47	3909.96	678.74
B2	1872.31	2145.02	288.99	5919.43	6140.72	232.02	3173.17	3347.33	196.21
B3	1849.60	4918.25	3145.30	5807.78	8186.62	2451.96	3428.98	5642.64	2246.87
B4	1826.34	2734.97	948.83	5804.11	6529.36	762.09	2862.31	3382.61	542.31
Number of Jobs									
10	47.46	100.12	61.25	141.57	186.58	58.16	224.84	277.36	57.20
30	477.06	851.24	398.71	1472.87	1761.30	303.68	995.98	1313.45	330.65
50	1361.84	2350.31	1030.14	4297.54	5078.58	822.49	2490.32	3220.64	746.29
70	2732.97	4607.72	1934.38	8652.75	10140.43	1516.27	4926.17	6222.52	1333.15
90	4639.20	7747.18	3183.48	14629.92	17048.64	2500.70	7286.34	9319.20	2112.86
Arrival parameter a									
0.25	4794.98	6544.49	1756.77	8186.63	9740.14	1575.86	6692.40	8125.71	1465.76
0.5	2936.46	4760.01	1838.90	7541.08	8824.66	1298.94	4888.83	6080.26	1191.66
0.75	1248.88	3080.65	1862.68	6177.81	7241.29	1107.18	3267.78	4225.96	986.35
1.25	178.39	904.49	813.47	4105.41	4794.23	741.15	680.01	1272.12	630.34
1.75	99.83	366.93	336.13	3183.72	3615.23	478.16	394.64	649.12	306.05
Process time									
P1	1640.11	2871.50	1275.95	5908.81	6919.60	1045.98	3101.55	3917.45	854.13
P2	2063.30	3391.13	1367.23	5769.05	6766.61	1034.54	3267.91	4223.82	977.93
Due Date									
D1	2117.34	3431.31	1328.22	6131.46	7160.10	1037.12	3431.46	4337.53	929.94
D2	2109.13	3440.86	1339.42	6121.87	7145.97	1047.61	3417.64	4342.29	924.37
D3	1548.17	2771.33	1307.65	5493.08	6468.65	1025.38	2906.36	3768.29	898.55
D4	1632.18	2881.76	1311.07	5609.31	6597.72	1050.91	2983.46	3834.43	911.27
Overall	1851.70	3131.31	1321.59	5838.93	6843.11	1040.26	3184.73	4070.64	916.03

Table 29. Total Flow Time Results (Stability)

Problem Parameters	Classical			Slack			Method 1		
	Estimate	Realized	Stability	Estimate	Realized	Stability	Estimate	Realized	Stability
Breakdown									
B1	2397.55	3306.12	908.57	6351.34	7055.97	714.96	3796.80	4468.65	678.74
B2	2417.06	2699.26	282.20	6449.37	6680.20	232.02	3690.68	3879.87	196.21
B3	2390.13	5541.98	3151.85	6334.86	8778.58	2451.96	3964.98	6172.51	2246.87
B4	2365.15	3306.39	941.25	6329.30	7086.32	762.09	3379.38	3937.72	542.31
Number of Jobs									
10	130.18	185.63	55.44	234.85	279.16	58.16	328.10	395.19	57.20
30	759.42	1153.90	394.48	1778.98	2080.71	303.68	1298.97	1628.34	330.65
50	1868.94	2903.80	1034.86	4822.88	5610.67	822.49	3012.99	3763.77	746.29
70	3503.39	5455.05	1951.66	9393.97	10923.47	1516.27	5659.12	6969.51	1333.15
90	5700.43	8868.82	3168.39	15600.42	18107.33	2500.70	8240.63	10316.63	2112.86
Arrival parameter a									
0.25	5498.51	7267.47	1768.96	8783.04	10344.66	1575.86	7307.55	8754.05	1465.76
0.5	3652.61	5502.95	1850.35	8122.12	9420.47	1298.94	5486.61	6688.68	1191.66
0.75	1844.46	3660.61	1816.14	6736.19	7837.53	1107.18	3839.50	4801.80	986.35
1.25	553.73	1377.96	824.24	4586.03	5324.64	741.15	1133.16	1743.58	630.34
1.75	413.05	758.19	345.14	3603.72	4074.06	478.16	772.98	1085.33	306.05
Process time									
P1	2181.38	3433.25	1251.87	6438.10	7490.46	1045.98	3629.27	4468.51	854.13
P2	2603.56	3993.62	1390.06	6294.34	7310.08	1034.54	3786.65	4760.87	977.93
Due Date									
D1	2399.42	3700.98	1301.55	6395.03	7420.25	1037.12	3695.49	4599.65	929.94
D2	2409.45	3741.23	1331.79	6405.11	7471.78	1047.61	3700.53	4611.37	924.37
D3	2390.20	3724.56	1334.37	6328.78	7359.83	1025.38	3729.74	4631.40	898.55
D4	2370.82	3686.97	1316.15	6335.95	7349.22	1050.91	3706.09	4616.33	911.27
Overall	2392.47	3713.44	1320.97	6366.22	7400.27	1040.26	3707.96	4614.69	916.03

Table 30. Makespan Results (Bicriterion)

Problem Parameters	Classical			Slack			Method 1		
	Estimate	Realized	Stability	Estimate	Realized	Stability	Estimate	Realized	Stability
Breakdown									
B1	348.23	377.29	1018.56	436.99	463.78	710.02	354.02	379.43	808.17
B2	348.31	357.42	311.58	437.31	446.05	226.25	349.68	357.54	244.95
B3	348.14	448.99	3507.81	437.68	525.05	2410.97	369.34	453.49	2384.19
B4	349.15	379.36	1056.89	437.02	465.44	748.44	355.64	380.11	696.82
Number of Jobs									
10	71.25	82.49	67.12	84.33	94.33	55.00	72.02	83.00	65.66
30	210.11	237.45	465.34	260.73	283.47	299.55	215.07	238.56	355.22
50	349.22	392.04	1190.43	437.36	474.99	795.37	357.08	392.54	847.54
70	487.04	544.53	2199.57	613.49	665.93	1521.52	499.50	547.72	1531.87
90	624.66	697.30	3446.11	790.35	856.68	2448.16	642.18	701.40	2367.38
Arrival parameter a									
0.25	276.47	338.49	2080.20	317.44	371.03	1528.31	281.84	337.58	1584.65
0.5	277.99	339.04	2070.08	358.98	405.50	1288.84	290.37	340.26	1393.33
0.75	284.74	343.24	1971.94	396.55	435.52	1103.28	304.65	347.45	1226.68
1.25	378.81	401.60	891.08	495.43	523.04	736.71	384.74	405.66	663.55
1.75	524.26	531.45	355.26	617.85	640.31	462.45	524.24	532.27	299.46
Process time									
P1	348.73	390.84	1487.95	442.26	481.51	1036.38	357.75	393.35	980.68
P2	348.18	390.69	1459.48	432.24	468.65	1011.46	356.58	391.94	1086.39
Due Date									
D1	348.91	391.66	1495.14	437.72	475.62	1030.24	357.02	392.71	1039.47
D2	348.23	390.93	1481.14	436.74	474.83	1029.74	356.69	391.64	1018.39
D3	348.19	390.70	1478.74	437.30	474.86	1002.15	357.52	393.20	1042.39
D4	348.50	389.77	1439.82	437.25	475.02	1033.55	357.46	393.02	1033.88
Overall	348.46	390.76	1473.71	437.25	475.08	1023.92	357.17	392.64	1033.53

Table 31. Total Tardiness Results (Bicriterion)

Problem Parameters	Classical			Slack			Method 1		
	Estimate	Realized	Stability	Estimate	Realized	Stability	Estimate	Realized	Stability
Breakdown									
B1	1845.35	2716.65	905.44	2625.74	3263.01	669.35	2048.63	2789.16	774.41
B2	1852.68	2119.50	282.54	2613.61	2795.59	196.93	1916.33	2139.94	239.43
B3	1834.82	4874.08	3117.74	2645.83	5088.26	2513.66	2654.67	4950.64	2372.74
B4	1852.24	2754.51	943.26	2586.75	3215.92	667.44	2165.41	2799.16	676.44
Number of Jobs									
10	47.70	99.61	59.85	60.57	107.03	53.00	47.20	94.77	56.95
30	468.44	849.52	407.72	673.25	985.62	335.55	531.50	850.07	345.11
50	1362.53	2356.58	1036.43	1935.03	2704.15	807.76	1570.60	2350.60	822.51
70	2771.91	4673.23	1961.31	3902.17	5340.13	1494.55	3253.02	4701.08	1508.66
90	4580.79	7601.98	3095.91	6518.89	8816.56	2368.35	5578.96	7852.10	2345.55
Arrival parameter a									
0.25	4776.21	6495.76	1727.27	4669.26	6362.93	1702.33	5094.32	6600.59	1516.12
0.5	2924.17	4728.07	1819.39	3619.27	4895.74	1298.64	3558.37	4881.21	1342.84
0.75	1262.02	3088.36	1858.15	2658.27	3698.16	1077.01	1922.91	3084.91	1199.90
1.25	172.74	905.80	820.31	1218.72	1807.47	657.78	287.37	908.68	695.90
1.75	96.24	362.93	336.10	924.38	1189.19	323.46	118.32	373.24	324.03
Process time									
P1	1638.12	2854.65	1261.57	2440.67	3326.22	926.25	1977.52	2901.69	966.46
P2	2054.42	3377.71	1362.92	2795.29	3855.17	1097.43	2415.00	3437.76	1065.05
Due Date									
D1	2109.93	3386.56	1290.11	2883.52	3873.82	1003.16	2460.31	3467.77	1021.88
D2	2121.49	3438.00	1324.47	2815.58	3820.49	1012.20	2435.76	3430.54	1003.48
D3	1534.12	2754.81	1306.22	2350.90	3288.05	1015.05	1906.88	2844.02	1022.28
D4	1619.56	2885.37	1328.18	2421.94	3380.42	1016.96	1982.08	2936.57	1015.39
Overall	1846.27	3116.18	1312.24	2617.98	3590.70	1011.84	2196.26	3169.73	1015.76

Table 32. Total Flow Time Results (Bicriterion)

Problem Parameters	Classical			Slack			Method 1		
	Estimate	Realized	Stability	Estimate	Realized	Stability	Estimate	Realized	Stability
Breakdown									
B1	2389.24	3314.06	924.82	3123.75	3778.06	654.31	2588.88	3357.20	768.33
B2	2371.22	2647.62	276.41	3140.95	3340.85	199.90	2439.77	2675.87	236.10
B3	2404.65	5597.43	3192.78	3153.49	5665.02	2511.53	3173.50	5533.00	2359.50
B4	2377.33	3316.80	939.48	3112.31	3795.30	682.98	2671.02	3347.92	676.90
Number of Jobs									
10	129.51	191.87	62.36	147.30	194.06	46.76	129.22	188.33	59.11
30	748.12	1152.80	404.69	950.01	1281.39	331.39	818.65	1163.35	344.70
50	1857.56	2903.60	1046.04	2423.41	3220.64	797.23	2076.91	2900.25	823.35
70	3516.37	5505.19	1988.83	4584.30	6065.19	1480.89	3984.13	5471.99	1487.86
90	5676.50	8841.43	3164.93	7558.11	9962.75	2404.64	6582.55	8918.57	2336.02
Arrival parameter a									
0.25	5488.90	7274.45	1785.54	5351.47	7049.50	1698.03	5739.66	7258.66	1519.00
0.5	3609.69	5460.73	1851.04	4212.03	5522.83	1310.80	4177.20	5524.38	1347.18
0.75	1877.06	3740.51	1863.45	3232.12	4300.61	1068.49	2515.34	3694.19	1178.85
1.25	542.87	1362.02	819.15	1601.21	2233.08	631.87	715.06	1421.96	706.90
1.75	409.52	757.19	347.66	1266.30	1618.01	351.71	444.21	743.31	299.10
Process time									
P1	2169.19	3444.00	1274.81	2944.45	3868.41	923.96	2480.53	3441.63	961.10
P2	2602.03	3993.96	1391.93	3320.80	4421.21	1100.40	2956.06	4015.37	1059.31
Due Date									
D1	2383.30	3708.58	1325.28	3137.89	4134.12	996.23	2726.94	3734.07	1007.14
D2	2378.10	3720.77	1342.67	3106.88	4113.44	1006.56	2743.21	3773.56	1030.35
D3	2395.98	3742.02	1346.04	3164.68	4181.37	1016.69	2706.13	3727.94	1021.81
D4	2385.05	3704.54	1319.49	3121.05	4150.30	1029.25	2696.89	3678.42	981.53
Overall	2385.61	3718.98	1333.37	3132.63	4144.81	1012.18	2718.29	3728.50	1010.21

Table 33. Makespan Results (Continuous Scheduling)

Problem Parameters	Classical			Slack			Method 1			Method 2		
	Estimate	Realized	Stability	Estimate	Realized	Stability	Estimate	Realized	Stability	Estimate	Realized	Stability
Breakdown												
B1	210.33	225.12	340.46	261.47	264.41	704.81	210.32	226.43	347.69	209.40	228.58	831.64
B2	209.92	214.52	97.06	261.00	251.95	598.42	210.10	214.56	101.70	210.37	215.48	577.23
B3	210.11	266.72	1195.57	260.18	287.10	1346.78	211.40	270.25	1211.90	210.33	278.28	1929.84
B4	210.17	224.89	315.15	260.46	247.99	998.57	209.95	225.59	318.10	210.39	228.08	1048.62
Number of Jobs												
10	71.76	77.80	36.56	84.10	89.01	47.18	71.39	79.63	46.26	71.24	80.69	75.47
30	210.74	235.33	403.09	261.54	267.01	647.72	210.46	235.34	415.57	209.91	238.25	882.19
50	347.89	385.30	1021.52	436.69	432.56	2041.53	349.47	387.65	1022.71	349.21	393.88	2332.84
Arrival parameter a												
0.25	166.06	198.62	661.24	188.44	214.44	707.20	166.23	199.59	670.68	165.68	196.33	1329.33
0.5	168.45	199.95	669.14	214.08	232.08	1000.63	168.50	201.46	650.48	168.18	202.65	1202.58
0.75	173.76	203.30	633.98	237.21	243.71	976.35	173.85	202.76	628.49	173.94	207.56	942.56
1.25	228.95	242.09	287.02	296.44	279.26	1007.04	230.16	244.28	324.40	229.21	250.15	802.27
1.75	313.44	320.10	183.91	367.73	344.82	869.50	313.46	322.94	200.20	313.60	331.34	1207.42
Process time												
P1	210.16	234.14	512.09	264.12	267.65	906.81	210.85	234.70	502.18	210.17	237.40	1095.67
P2	210.10	231.48	462.03	257.43	258.08	917.48	210.02	233.71	487.52	210.07	237.81	1097.99
Due Date												
D1	210.81	232.95	491.40	260.56	261.72	921.02	210.73	234.36	485.63	210.56	236.07	1069.76
D2	209.54	232.91	477.32	260.96	262.46	905.39	210.62	233.83	499.53	210.12	237.70	1084.58
D3	209.71	232.44	489.01	260.43	261.92	928.70	210.07	234.44	516.68	209.62	238.19	1122.88
D4	210.48	232.94	490.50	261.17	265.35	893.48	210.35	234.19	477.56	210.19	238.47	1110.11
Overall	210.13	232.81	487.06	260.78	262.86	912.15	210.44	234.21	494.85	210.12	237.61	1096.83

Table 34. Total Tardiness Results (Continuous Scheduling)

Problem Parameters	Classical			Slack			Method 1			Method 2		
	Estimate	Realized	Stability	Estimate	Realized	Stability	Estimate	Realized	Stability	Estimate	Realized	Stability
Breakdown												
B1	618.60	866.15	315.67	846.58	972.41	375.02	664.65	903.88	312.54	674.59	936.21	314.68
B2	616.78	683.24	129.10	852.27	760.44	235.94	638.18	704.68	146.14	642.95	710.07	132.25
B3	629.91	1455.63	912.25	860.52	1526.51	924.01	811.82	1534.58	852.00	669.09	1535.97	946.98
B4	630.78	839.95	299.37	838.89	848.51	382.64	678.29	860.05	293.54	640.63	865.75	310.19
Number of Jobs												
10	47.74	85.05	49.11	58.74	84.12	37.27	46.41	80.71	44.67	49.53	79.05	39.06
30	470.66	746.86	321.23	635.14	800.64	365.36	508.21	766.58	316.40	494.93	781.64	334.95
50	1353.66	2051.83	871.95	1854.81	2196.15	1035.58	1540.08	2155.10	842.09	1425.98	2175.31	904.07
Arrival parameter a												
0.25	1569.75	2047.30	534.77	1567.04	2111.62	574.86	1619.24	2105.02	543.30	1617.77	2122.68	546.97
0.5	962.29	1425.49	605.21	1110.37	1495.89	505.33	1088.75	1512.32	590.31	1020.22	1492.41	575.85
0.75	440.91	880.92	530.46	821.81	971.08	533.04	586.91	926.33	478.94	491.80	959.90	566.19
1.25	90.18	298.30	264.04	421.44	353.69	444.20	129.22	311.45	258.53	96.16	326.07	290.55
1.75	56.97	154.21	136.00	327.16	202.56	339.59	67.05	148.86	134.20	58.11	158.95	150.57
Process time												
P1	548.97	845.59	383.68	800.49	930.78	448.19	628.44	888.16	373.08	585.07	892.93	392.34
P2	699.06	1076.90	444.51	898.64	1123.16	510.62	768.03	1113.43	429.03	728.56	1131.07	459.71
Due Date												
D1	763.10	1103.82	424.87	970.72	1173.07	496.51	857.53	1165.81	414.33	802.74	1154.92	418.93
D2	750.36	1126.06	458.80	985.93	1194.48	489.80	812.41	1139.97	427.60	780.92	1161.85	451.43
D3	474.48	776.79	374.49	694.58	861.28	489.78	533.12	827.41	392.52	496.55	843.18	422.09
D4	508.13	838.31	398.23	747.03	879.05	441.51	589.88	869.99	369.78	547.05	888.06	411.66
Overall	624.02	961.24	414.10	849.57	1026.97	479.40	698.24	1000.80	401.06	656.81	1012.00	426.03

Table 35. Total Flow Time Results (Continuous Scheduling)

Problem Parameters	Classical			Slack			Method 1			Method 2		
	Estimate	Realized	Stability	Estimate	Realized	Stability	Estimate	Realized	Stability	Estimate	Realized	Stability
Breakdown												
B1	911.05	1171.03	330.27	1141.24	1258.22	363.57	952.98	1219.73	359.54	907.19	1194.38	324.17
B2	912.13	980.27	165.71	1144.05	1058.04	236.06	913.67	974.14	155.53	913.57	986.18	140.53
B3	901.23	1807.14	974.52	1125.72	1849.58	934.00	1122.48	1905.64	900.64	904.82	1856.69	978.32
B4	928.85	1154.14	350.51	1122.95	1148.28	390.68	987.16	1171.96	332.75	909.34	1147.29	313.73
Number of Jobs												
10	129.65	179.19	52.90	143.78	183.10	45.11	128.91	173.59	47.18	128.03	163.72	37.12
30	750.96	1039.47	319.95	918.27	1120.27	380.83	801.34	1071.64	320.27	746.12	1069.64	347.55
50	1859.34	2615.77	992.90	2338.42	2682.21	1017.29	2051.96	2708.37	943.90	1852.03	2655.05	932.90
Arrival parameter a												
0.25	1957.58	2467.99	597.66	1924.43	2464.32	604.02	1988.68	2500.66	595.07	1927.52	2494.71	621.91
0.5	1305.10	1756.20	664.94	1445.45	1804.43	501.25	1447.65	1873.45	669.34	1294.89	1809.39	600.09
0.75	760.96	1246.63	613.00	1134.87	1325.12	541.35	918.67	1258.23	510.99	766.77	1241.67	564.93
1.25	304.24	565.14	277.97	637.53	631.81	440.05	358.37	575.46	264.51	312.90	558.63	266.76
1.75	238.70	354.76	122.70	525.17	416.95	318.70	256.98	381.53	145.68	241.55	376.28	142.25
Process time												
P1	844.20	1159.30	412.14	1064.27	1208.85	437.99	902.82	1181.71	393.98	824.42	1158.13	390.88
P2	982.43	1396.99	498.37	1202.71	1448.20	524.16	1085.32	1454.02	480.25	993.04	1434.14	487.49
Due Date												
D1	917.84	1278.28	458.75	1146.52	1348.58	490.35	993.65	1325.07	454.44	915.43	1288.20	420.24
D2	916.28	1268.89	445.66	1131.13	1312.78	470.27	997.30	1304.14	422.69	914.58	1309.92	447.55
D3	915.66	1288.60	455.22	1141.38	1326.77	477.12	984.19	1296.17	425.98	894.77	1287.40	447.94
D4	903.48	1276.81	461.38	1114.93	1325.98	486.56	1001.14	1346.08	445.35	910.13	1299.02	441.02
Overall	913.31	1278.14	455.25	1133.49	1328.53	481.08	994.07	1317.87	437.12	908.73	1296.14	439.19

SCHEDULE.C

```
#include <ctype.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define GAMMA_07 1.298055F
#define GAMMA_14 0.887264F
#define TAIL1    0.00455F
#define TAIL2    3.02769F

/* type definitions*/

typedef struct
        {
           int id;
           double completion_time;
        } schedule;


typedef struct
        {
           int id;
           int arrival_time;
           int process_time;
           int due_date;
        } job;

typedef struct
        {
           int first_job_id;
           int first_job_from_position;
           int first_job_to_position;
           int second_job_id;
           int second_job_from_position;
           int second_job_to_position;
        } movetype;


typedef struct node
        {
           int job_id;
           int to_position;
           int remaining_tenure;
           struct node *next;
        } tabu;
```

```
typedef struct
        {
           movetype moves;
           job *sequence;
           double obj_value;
        } element;


/* global variables */

const char *program_name;       /* name of this program                        */
FILE *f, *g;                    /* input and output file handlers              */
int nremainingjobs;             /* number of remaining jobs                    */
int breakdown;                  /* breakdown scheme                            */
int tenure;                     /* tenure                                      */
int objective;                  /* objective function to be used               */
int performance_measure;        /* performance measure to be used              */
double best_obj;                /* best objective value                        */
double current_obj;             /* current objective value                     */
double (*obj_array[14])(job *, int weight); /* objective function table       */
int verbose = 0;                /* Shall we reveal details?                    */
int continuous = 0;             /* Reschedule after m/c breakdown?             */
int improve = 0;                /* Shall we prompt when current best improves  */
double lambda = 0;              /* scale parameter of the gamma distribution   */
tabu *tabu_list = NULL;         /* points to the head of the tabu list         */
tabu *tail = NULL;              /* points to the last element in the tabu list */
job *jobs;                      /* job list                                    */
element *neighbourhood;         /* current solutions neighbourhood             */
job *current_solution;          /* current solution                            */
job *best_solution;             /* best solution found so far                  */
int iIter, counter;             /* number of current iteration                 */
double *cdf;                    /* cdf values for gamma distribution           */
int interval = 0;
int last_improved = 0;
double mean_time_to_failure = 0;
double mean_repair_duration = 0;
double stabil;
int weight;

/* function decleration of external getopt() */
int getopt(int argc, char * const *argv, const char *optstring);

double power(double x, double y)
{
   if (x == 0) return 0;
   else if (y == 0) return 1;
   else return pow(x,y);
}
```

```
int fread_number (FILE * fp)
{
  int number;
  int sign;
  int c;

  do
    {
      c = getc (fp);
    }
  while (isspace (c));
  number = 0;
  sign = 0;
  if (c == '-')
    {
      sign = 1;
      c = getc (fp);
    }
  while (isdigit (c))
    {
      number = number * 10 + c - '0';
      c = getc (fp);
    }
  if (sign)
    number = 0 - number;
  return number;
}

double integrate(double (*f)(double), double lower, double upper)
{
   double increment = 0.01;
   double sum = 0;
   double index;

   for (index = lower; index < upper; index += increment)
      sum += (((f(index) + f(index + increment)) / 2.0) * increment);
   return sum;
}

double failure_dist(double x)
{
   return (power(lambda, 0.7) * power(x, -0.3) * exp(-lambda * x) / GAMMA_07);
}

double repair_dist(double x)
{
   return (power(lambda, 1.4) * power(x, 0.4) * exp(-lambda * x) / GAMMA_14);
}
```

```
double makespan(job *sequence, int r)
{
   int *completion_times;
   int i;
   int cmax;

   completion_times = (int *) malloc(nremainingjobs * sizeof(int));

   completion_times[0] = sequence[0].arrival_time + sequence[0].process_time;
   for (i = 1; i < nremainingjobs; i++)
      completion_times[i] = sequence[i].process_time +
                            ((completion_times[i - 1] < sequence[i].arrival_time) ?
                            sequence[i].arrival_time :
                            completion_times[i - 1]);

   cmax = completion_times[nremainingjobs - 1];
   free(completion_times);
   return cmax;
}

double tardiness(job *sequence, int r)
{
   int *completion_times;
   int i;
   int tardy;
   int sum = 0;

   completion_times = (int *) malloc(nremainingjobs * sizeof(int));

   completion_times[0] = sequence[0].arrival_time + sequence[0].process_time;
   for (i = 1; i < nremainingjobs; i++)
      completion_times[i] = sequence[i].process_time +
                            ((completion_times[i - 1] < sequence[i].arrival_time) ?
                            sequence[i].arrival_time :
                            completion_times[i - 1]);

   for (i = 0; i < nremainingjobs; i++)
      sum += ((tardy = completion_times[i] - sequence[i].due_date) > 0 ? tardy : 0);
   free(completion_times);
   return sum;
}

double flowtime(job *sequence, int r)
{
   int *completion_times;
   int i;
   int sum = 0;

   completion_times = (int *) malloc(nremainingjobs * sizeof(int));

   completion_times[0] = sequence[0].arrival_time + sequence[0].process_time;
   for (i = 1; i < nremainingjobs; i++)
      completion_times[i] = sequence[i].process_time +
                            ((completion_times[i - 1] < sequence[i].arrival_time) ?
```

```
                              sequence[i].arrival_time :
                              completion_times[i - 1]);

   for (i = 0; i < nremainingjobs; i++)
      sum += (completion_times[i] - sequence[i].arrival_time);
   free(completion_times);
   return sum;
}

int get_total_idle_time(double *start_times, double *completion_times, int index, int
job)
{
   int i;
   double sum = 0;

   for (i = index; i < job; i++)
      sum += ((completion_times[i] < start_times[i + 1]) ?
             (start_times[i + 1] - completion_times[i]) : 0);
   return sum;
}

double method1_stability(job *sequence, int r)
{
   double *completion_times;
   double *completion_times1;
   int i;
   double tnow = 0;
   int job_completed;
   double busy_time = 0;
   double remaining_process;
   double mean_busy_time;
   double sum = 0;
   double tail1, tail2;

   tail1 = (mean_time_to_failure / 0.7) * TAIL1;
   tail2 = (mean_time_to_failure / 0.7) * TAIL2;
   mean_busy_time = 0.6 * tail1 + 0.4 * tail2;
   completion_times = (double *) malloc(nremainingjobs * sizeof(double));
   completion_times1 = (double *) malloc(nremainingjobs * sizeof(double));

   completion_times[0] = sequence[0].arrival_time + sequence[0].process_time;
   for (i = 1; i < nremainingjobs; i++)
      completion_times[i] = sequence[i].process_time +
                              ((completion_times[i - 1] < sequence[i].arrival_time) ?
                              sequence[i].arrival_time :
                              completion_times[i - 1]);

   tnow = sequence[0].arrival_time;
   for (i = 0; i < nremainingjobs; i++)
   {
      job_completed = 0;
      remaining_process = sequence[i].process_time;
      if (busy_time == mean_busy_time)
      {
```

```
         tnow += mean_repair_duration;
         busy_time = 0;
      }
      if (tnow < sequence[i].arrival_time) tnow = sequence[i].arrival_time;
      do
      {
         if (remaining_process > mean_busy_time - busy_time)
         {
             tnow += (mean_repair_duration + mean_busy_time - busy_time);
             remaining_process -= mean_busy_time - busy_time;
             busy_time = 0;
         }
         else
         {
            busy_time += remaining_process;
            tnow += remaining_process;
            job_completed = 1;
         }
      }
      while (!job_completed);
      completion_times1[i] = tnow;
   }
   for (i = 0; i < nremainingjobs; i++)
      sum += (completion_times[i] < completion_times1[i] ?
                 completion_times1[i] - completion_times[i] :
                 completion_times[i] - completion_times1[i] );
   free(completion_times);
   free(completion_times1);
   return (sum);
}

double method2_stability(job *sequence, int r)
{
   double *start_times;
   double *completion_times;
   double *stability_values;
   double *temp_completion_times;
   double *probabilities;
   int i, j, k;
   double idle;
   double sum = 0;
   int total_busy_time_before;

   start_times = (double *) malloc (nremainingjobs * sizeof(double));
   completion_times = (double *) malloc (nremainingjobs * sizeof(double));
   stability_values = (double *) malloc (nremainingjobs * sizeof(double));
   temp_completion_times = (double *) malloc (nremainingjobs * sizeof(double));
   probabilities = (double *) malloc (nremainingjobs * sizeof(double));
   start_times[0] = sequence[0].arrival_time;
   completion_times[0] = start_times[0] + sequence[0].process_time;
   for (i = 1; i < nremainingjobs; i++)
   {
      start_times[i] = sequence[i].arrival_time < completion_times[i-1] ?
                       completion_times[i-1] :
```

```
                       sequence[i].arrival_time;
      completion_times[i] = start_times[i] + sequence[i].process_time;
   }
   for (i = 0; i < nremainingjobs; i++)
   {
      sum = 0;
      for (j = 0; j < nremainingjobs; j++)
      {
         if (j < i) temp_completion_times[j] = completion_times[j];
         else temp_completion_times[j] = completion_times[j] +
              ((idle = get_total_idle_time(start_times, completion_times, i, j)) >
              mean_repair_duration ? 0 : (mean_repair_duration - idle));
      }
      for (k = 0; k < nremainingjobs; k++)
         sum += (temp_completion_times[k] - completion_times[k]);
      stability_values[i] = sum;
      total_busy_time_before = 0;
      for (k = 0; k < i; k++) total_busy_time_before += sequence[k].process_time;
      probabilities[i] = cdf[total_busy_time_before + sequence[i].process_time]
                       - cdf[total_busy_time_before];
   }

   for (i = 0; i < nremainingjobs; i++)
      sum += (probabilities[i] * stability_values[i]);

   free (start_times);
   free (completion_times);
   free (stability_values);
   free (temp_completion_times);
   free (probabilities);
   return (sum);
}

double slack_makespan (job *sequence, int r)
{
   int i, j;
   int *completion_times;
   int *start_times;
   int sum, slack = 0;

   start_times = (int *) malloc (nremainingjobs * sizeof(int));
   completion_times = (int *) malloc (nremainingjobs * sizeof(int));
   start_times[0] = sequence[0].arrival_time;
   completion_times[0] = start_times[0] + sequence[0].process_time;
   for (i = 1; i < nremainingjobs; i++)
   {
      start_times[i] = sequence[i].arrival_time < completion_times[i-1] ?
                       completion_times[i-1] :
                       sequence[i].arrival_time;
      completion_times[i] = start_times[i] + sequence[i].process_time;
   }
```

```
    for (i = 0; i < nremainingjobs; i++)
    {
       sum = 0;
       for (j =  i + 1; j < nremainingjobs; j++) sum += sequence[j].process_time;
       slack += (completion_times[nremainingjobs -1] - sum - completion_times[i]);
    }
    free(start_times);
    free(completion_times);
    return (r * (makespan(sequence, r) - slack) / 100.0 + (r - 100) * slack / 100.0);
}

double slack_tardiness (job *sequence, int r)
{
    int i, j;
    int *completion_times;
    int *start_times;
    int sum, slack = 0;

    start_times = (int *) malloc (nremainingjobs * sizeof(int));
    completion_times = (int *) malloc (nremainingjobs * sizeof(int));
    start_times[0] = sequence[0].arrival_time;
    completion_times[0] = start_times[0] + sequence[0].process_time;
    for (i = 1; i < nremainingjobs; i++)
    {
       start_times[i] = sequence[i].arrival_time < completion_times[i-1] ?
                        completion_times[i-1] :
                        sequence[i].arrival_time;
       completion_times[i] = start_times[i] + sequence[i].process_time;
    }
    for (i = 0; i < nremainingjobs; i++)
    {
       sum = 0;
       for (j =  i + 1; j < nremainingjobs; j++) sum += sequence[j].process_time;
       slack += (completion_times[nremainingjobs -1] - sum - completion_times[i]);
    }
    free(start_times);
    free(completion_times);
    return (r * (tardiness(sequence, r) - slack) / 100.0 + (r - 100) * slack / 100.0);
}

double slack_flowtime (job *sequence, int r)
{
    int i, j;
    int *completion_times;
    int *start_times;
    int sum, slack = 0;

    start_times = (int *) malloc (nremainingjobs * sizeof(int));
    completion_times = (int *) malloc (nremainingjobs * sizeof(int));
    start_times[0] = sequence[0].arrival_time;
    completion_times[0] = start_times[0] + sequence[0].process_time;
```

```
   for (i = 1; i < nremainingjobs; i++)
   {
      start_times[i] = sequence[i].arrival_time < completion_times[i-1] ?
                       completion_times[i-1] :
                       sequence[i].arrival_time;
      completion_times[i] = start_times[i] + sequence[i].process_time;
   }
   for (i = 0; i < nremainingjobs; i++)
   {
      sum = 0;
      for (j =  i + 1; j < nremainingjobs; j++) sum += sequence[j].process_time;
      slack += (completion_times[nremainingjobs -1] - sum - completion_times[i]);
   }
   free(start_times);
   free(completion_times);
   return (r * (flowtime(sequence, r) - slack) / 100.0 + (r - 100) * slack / 100.0);
}

double method1_makespan(job *sequence, int r)
{
   double *completion_times;
   int i;
   double tnow = 0;
   int job_completed;
   double busy_time = 0;
   double remaining_process;
   double mean_busy_time;
   double cmax;
   double tail1, tail2;

   tail1 = (mean_time_to_failure / 0.7) * TAIL1;
   tail2 = (mean_time_to_failure / 0.7) * TAIL2;
   mean_busy_time = 0.6 * tail1 + 0.4 * tail2;
   completion_times = (double *) malloc(nremainingjobs * sizeof(double));

   tnow = sequence[0].arrival_time;
   for (i = 0; i < nremainingjobs; i++)
   {
      job_completed = 0;
      remaining_process = sequence[i].process_time;
      if (busy_time == mean_busy_time)
      {
         tnow += mean_repair_duration;
         busy_time = 0;
      }
      if (tnow < sequence[i].arrival_time) tnow = sequence[i].arrival_time;
      do
      {
         if (remaining_process > mean_busy_time - busy_time)
         {
             tnow += (mean_repair_duration + mean_busy_time - busy_time);
             remaining_process -= mean_busy_time - busy_time;
             busy_time = 0;
         }
```

```
        else
        {
           busy_time += remaining_process;
           tnow += remaining_process;
           job_completed = 1;
        }
     }
     while (!job_completed);
     completion_times[i] = tnow;
  }
  cmax = completion_times[nremainingjobs - 1];
  free(completion_times);
  return (r * cmax / 100.0 +  (100 - r) * method1_stability(sequence, r) / 100.0);
}

double method1_tardiness(job *sequence, int r)
{
  double *completion_times;
  int i;
  double tnow = 0;
  int job_completed;
  double busy_time = 0;
  double remaining_process;
  double sum = 0, tardy;
  double mean_busy_time;
  double tail1, tail2;

  tail1 = (mean_time_to_failure / 0.7) * TAIL1;
  tail2 = (mean_time_to_failure / 0.7) * TAIL2;
  mean_busy_time = 0.6 * tail1 + 0.4 * tail2;
  completion_times = (double *) malloc(nremainingjobs * sizeof(double));

  tnow = sequence[0].arrival_time;
  for (i = 0; i < nremainingjobs; i++)
  {
     job_completed = 0;
     remaining_process = sequence[i].process_time;
     if (busy_time == mean_busy_time)
     {
        tnow += mean_repair_duration;
        busy_time = 0;
     }
     if (tnow < sequence[i].arrival_time) tnow = sequence[i].arrival_time;
     do
     {
        if (remaining_process > mean_busy_time - busy_time)
        {
            tnow += (mean_repair_duration + mean_busy_time - busy_time);
            remaining_process -= mean_busy_time - busy_time;
            busy_time = 0;
        }
```

```
         else
         {
            busy_time += remaining_process;
            tnow += remaining_process;
            job_completed = 1;
         }
      }
      while (!job_completed);
      completion_times[i] = tnow;
   }
   for (i = 0; i < nremainingjobs; i++)
      sum += ((tardy = completion_times[i] - sequence[i].due_date) > 0 ? tardy : 0);
   free(completion_times);
   return (r * sum / 100.0 + (100 - r) * method1_stability(sequence, r) / 100.0);
}

double method1_flowtime(job *sequence, int r)
{
   double *completion_times;
   int i;
   double tnow = 0;
   int job_completed;
   double busy_time = 0;
   double remaining_process;
   double sum = 0;
   double mean_busy_time;
   double tail1, tail2;

   tail1 = (mean_time_to_failure / 0.7) * TAIL1;
   tail2 = (mean_time_to_failure / 0.7) * TAIL2;
   mean_busy_time = 0.6 * tail1 + 0.4 * tail2;
   completion_times = (double *) malloc(nremainingjobs * sizeof(double));

   tnow = sequence[0].arrival_time;
   for (i = 0; i < nremainingjobs; i++)
   {
      job_completed = 0;
      remaining_process = sequence[i].process_time;
      if (busy_time == mean_busy_time)
      {
         tnow += mean_repair_duration;
         busy_time = 0;
      }
      if (tnow < sequence[i].arrival_time) tnow = sequence[i].arrival_time;
      do
      {
         if (remaining_process > mean_busy_time - busy_time)
         {
             tnow += (mean_repair_duration + mean_busy_time - busy_time);
             remaining_process -= mean_busy_time - busy_time;
             busy_time = 0;
         }
```

```
            else
            {
               busy_time += remaining_process;
               tnow += remaining_process;
               job_completed = 1;
            }
         }
         while (!job_completed);
         completion_times[i] = tnow;
      }
      for (i = 0; i < nremainingjobs; i++)
         sum += (completion_times[i] - sequence[i].arrival_time);
      free(completion_times);
      return (r * sum / 100.0 +  (100 - r) * method1_stability(sequence, r) / 100.0);
   }

   double method2_makespan(job *sequence, int r)
   {
      double *start_times;
      double *completion_times;
      double *makespan_values;
      double *temp_completion_times;
      double *probabilities;
      int i, j, k;
      double idle;
      double sum = 0;
      int total_busy_time_before;

      start_times = (double *) malloc (nremainingjobs * sizeof(double));
      completion_times = (double *) malloc (nremainingjobs * sizeof(double));
      makespan_values = (double *) malloc (nremainingjobs * sizeof(double));
      temp_completion_times = (double *) malloc (nremainingjobs * sizeof(double));
      probabilities = (double *) malloc (nremainingjobs * sizeof(double));
      start_times[0] = sequence[0].arrival_time;
      completion_times[0] = start_times[0] + sequence[0].process_time;
      for (i = 1; i < nremainingjobs; i++)
      {
         start_times[i] = sequence[i].arrival_time < completion_times[i-1] ?
                          completion_times[i-1] :
                          sequence[i].arrival_time;
         completion_times[i] = start_times[i] + sequence[i].process_time;
      }
      for (i = 0; i < nremainingjobs; i++)
      {
         sum = 0;
         for (j = 0; j < nremainingjobs; j++)
         {
            if (j < i) temp_completion_times[j] = completion_times[j];
            else temp_completion_times[j] = completion_times[j] +
                 ((idle = get_total_idle_time(start_times, completion_times, i, j)) >
                 mean_repair_duration ? 0 : (mean_repair_duration - idle));
         }
         makespan_values[i] = temp_completion_times[nremainingjobs - 1];
         total_busy_time_before = 0;
```

```
      for (k = 0; k < i; k++) total_busy_time_before += sequence[k].process_time;
      probabilities[i] = cdf[total_busy_time_before + sequence[i].process_time]
                         - cdf[total_busy_time_before];
   }

   for (i = 0; i < nremainingjobs; i++)
      sum += (probabilities[i] * makespan_values[i]);

   free (start_times);
   free (completion_times);
   free (makespan_values);
   free (temp_completion_times);
   free (probabilities);
   return (r * sum / 100.0 + (100 - r) * method2_stability(sequence, r) / 100.0);
}

double method2_tardiness (job *sequence, int r)
{
   double *start_times;
   double *completion_times;
   double *tardiness_values;
   double *temp_completion_times;
   double *probabilities;
   int i, j, k;
   double tardy, idle, sum;
   int total_busy_time_before;

   start_times = (double *) malloc (nremainingjobs * sizeof(double));
   completion_times = (double *) malloc (nremainingjobs * sizeof(double));
   tardiness_values = (double *) malloc (nremainingjobs * sizeof(double));
   temp_completion_times = (double *) malloc (nremainingjobs * sizeof(double));
   probabilities = (double *) malloc (nremainingjobs * sizeof(double));
   start_times[0] = sequence[0].arrival_time;
   completion_times[0] = start_times[0] + sequence[0].process_time;
   for (i = 1; i < nremainingjobs; i++)
   {
      start_times[i] = sequence[i].arrival_time < completion_times[i-1] ?
                         completion_times[i-1] :
                         sequence[i].arrival_time;
      completion_times[i] = start_times[i] + sequence[i].process_time;
   }
   for (i = 0; i < nremainingjobs; i++)
   {
      tardy = sum = 0;
      for (j = 0; j < nremainingjobs; j++)
      {
         if (j < i) temp_completion_times[j] = completion_times[j];
         else temp_completion_times[j] = completion_times[j] +
              ((idle = get_total_idle_time(start_times, completion_times, i, j)) >
              mean_repair_duration ? 0 : (mean_repair_duration - idle));
      }
      for (k = 0; k < nremainingjobs; k++)
         sum += ((tardy = temp_completion_times[k] - sequence[k].due_date) > 0 ? tardy
0);
```

```
      tardiness_values[i] = sum;
      total_busy_time_before = 0;
      for (k = 0; k < i; k++) total_busy_time_before += sequence[k].process_time;
      probabilities[i] = cdf[total_busy_time_before + sequence[i].process_time]
                       - cdf[total_busy_time_before];
   }
   sum = 0;
   for (i = 0; i < nremainingjobs; i++)
      sum += (probabilities[i] * tardiness_values[i]);
   free (start_times);
   free (completion_times);
   free (tardiness_values);
   free (temp_completion_times);
   free (probabilities);
   return (r * sum / 100.0 +  (100 - r) * method2_stability(sequence, r) / 100.0);
}

double method2_flowtime(job *sequence, int r)
{
   double *start_times;
   double *completion_times;
   double *flowtime_values;
   double *temp_completion_times;
   double *probabilities;
   int i, j, k;
   double idle;
   double sum = 0;
   int total_busy_time_before;

   start_times = (double *) malloc (nremainingjobs * sizeof(double));
   completion_times = (double *) malloc (nremainingjobs * sizeof(double));
   flowtime_values = (double *) malloc (nremainingjobs * sizeof(double));
   temp_completion_times = (double *) malloc (nremainingjobs * sizeof(double));
   probabilities = (double *) malloc (nremainingjobs * sizeof(double));
   start_times[0] = sequence[0].arrival_time;
   completion_times[0] = start_times[0] + sequence[0].process_time;
   for (i = 1; i < nremainingjobs; i++)
   {
      start_times[i] = sequence[i].arrival_time < completion_times[i-1] ?
                       completion_times[i-1] :
                       sequence[i].arrival_time;
      completion_times[i] = start_times[i] + sequence[i].process_time;
   }
   for (i = 0; i < nremainingjobs; i++)
   {
      sum = 0;
      for (j = 0; j < nremainingjobs; j++)
      {
         if (j < i) temp_completion_times[j] = completion_times[j];
         else temp_completion_times[j] = completion_times[j] +
               ((idle = get_total_idle_time(start_times, completion_times, i, j)) >
               mean_repair_duration ? 0 : (mean_repair_duration - idle));
      }
      for (k = 0; k < nremainingjobs; k++)
```

```
         sum += (temp_completion_times[k] - sequence[k].arrival_time);
      flowtime_values[i] = sum;
      total_busy_time_before = 0;
      for (k = 0; k < i; k++) total_busy_time_before += sequence[k].process_time;
      probabilities[i] = cdf[total_busy_time_before + sequence[i].process_time]
                       - cdf[total_busy_time_before];
   }

   for (i = 0; i < nremainingjobs; i++)
      sum += (probabilities[i] * flowtime_values[i]);

   free (start_times);
   free (completion_times);
   free (flowtime_values);
   free (temp_completion_times);
   free (probabilities);
   return (r * sum / 100.0 +  (100 - r) * method2_stability(sequence, r) / 100.0);
}

double get_failure()
{
   double b, e, u1, u2, p, y, x;
   e = exp(1);
   b = (e + 0.7) / e;
   for (;;)
   {
      u1 = (double) rand() / RAND_MAX;
      p = b * u1;
      if ( p > 1)
      {
         y = -log((b - p) / 0.7);
         u2 = (double) rand() / RAND_MAX;
         if (u2 <= pow (y, -0.3))
         {
            x = y;
            break;
         }
      }
      else
      {
         y = pow(p, 1.0 / 0.7);
         u2 = (double) rand() / RAND_MAX;
         if (u2 <= exp(-y))
         {
            x = y;
            break;
         }
      }
   }
   return ((mean_time_to_failure / 0.7) * x);
}
```

```
double get_repair()
{
   double a, b, q, teta, d, u1, u2, v, y, z, w, x;

   a = 1.0 / sqrt(1.8);
   b = 1.4 - log(4);
   q = 1.4 + 1.0 / a;
   teta = 4.5;
   d = 1 + log(teta);

   for (;;)
   {
      u1 = (double) rand() / RAND_MAX;
      u2 = (double) rand() / RAND_MAX;
      v = a * log(u1 / (1 - u1));
      y = 1.4 * exp(v);
      z = u1 * u1 * u2;
      w = b + q * v - y;
      if (w + d - teta * z >= 0)
      {
         x = y;
         break;
      }
      else
      {
         if (w >= log (z))
         {
            x = y;
            break;
         }
      }
   }
   return ((mean_repair_duration / 1.4) * x);
}

double simulate_makespan(job *sequence)
{
   double stability_values[5];
   double makespan_values[5];
   double *completion_times;
   double *completion_times1;
   int i, counter;
   double tnow;
   int job_completed;
   double busy_time;
   double remaining_process;
   double cmax = 0;
   double next_failure;

   completion_times = (double *) malloc(nremainingjobs * sizeof(double));
   completion_times1 = (double *) malloc(nremainingjobs * sizeof(double));
```

```
for (counter = 0; counter < 5; counter++)
{
   busy_time = 0;
   next_failure = get_failure();
   tnow = sequence[0].arrival_time;
   for (i = 0; i < nremainingjobs; i++)
   {
      job_completed = 0;
      remaining_process = sequence[i].process_time;
      if (busy_time == next_failure)
      {
         tnow += get_repair();
         busy_time = 0;
         next_failure = get_failure();
      }
      if (tnow < sequence[i].arrival_time) tnow = sequence[i].arrival_time;
      do
      {
         if (remaining_process > next_failure - busy_time)
         {
             tnow += (get_repair() + next_failure - busy_time);
             remaining_process -= (next_failure - busy_time);
             busy_time = 0;
             next_failure = get_failure();
         }
         else
         {
            busy_time += remaining_process;
            tnow += remaining_process;
            job_completed = 1;
         }
      }
      while (!job_completed);
      completion_times[i] = tnow;
   }
   completion_times1[0] = sequence[0].process_time + sequence[0].arrival_time;
   for (i = 1; i < nremainingjobs; i++)
   {
      completion_times1[i] = sequence[i].process_time +
                             (sequence[i].arrival_time < completion_times1[i-1] ?
                              completion_times1[i-1] : sequence[i].arrival_time);
   }
   stability_values[counter] = 0;
   for (i = 0; i < nremainingjobs; i++)
      stability_values[counter] += (completion_times[i] < completion_times1[i] ?
                   completion_times1[i] - completion_times[i] :
                   completion_times[i] - completion_times1[i] );

   makespan_values[counter] = completion_times[nremainingjobs - 1];
}
stabil = 0;
```

```
   for (counter = 0; counter < 5; counter++)
   {
      stabil += stability_values[counter];
      cmax += makespan_values[counter];
   }
   free(completion_times);
   free(completion_times1);
   stabil = stabil / 5.0;
   return (cmax / 5.0);
}

double simulate_tardiness(job *sequence)
{
   double stability_values[5];
   double tardiness_values[5];
   double *completion_times;
   double *completion_times1;
   int i, counter;
   double tnow;
   int job_completed;
   double busy_time;
   double remaining_process;
   double sum, tardy;
   double next_failure;

   completion_times = (double *) malloc(nremainingjobs * sizeof(double));
   completion_times1 = (double *) malloc(nremainingjobs * sizeof(double));
   for (counter = 0; counter < 5; counter++)
   {
      busy_time = 0;
      sum = 0;
      next_failure = get_failure();
      tnow = sequence[0].arrival_time;
      for (i = 0; i < nremainingjobs; i++)
      {
         job_completed = 0;
         remaining_process = sequence[i].process_time;
         if (busy_time == next_failure)
         {
            tnow += get_repair();
            busy_time = 0;
            next_failure = get_failure();
         }
         if (tnow < sequence[i].arrival_time) tnow = sequence[i].arrival_time;
         do
         {
            if (remaining_process > next_failure - busy_time)
            {
                tnow += (get_repair() + next_failure - busy_time);
                remaining_process -= (next_failure - busy_time);
                busy_time = 0;
                next_failure = get_failure();
            }
```

```
            else
            {
               busy_time += remaining_process;
               tnow += remaining_process;
               job_completed = 1;
            }
         }
         while (!job_completed);
         completion_times[i] = tnow;
      }
      completion_times1[0] = sequence[0].process_time + sequence[0].arrival_time;
      for (i = 1; i < nremainingjobs; i++)
      {
         completion_times1[i] = sequence[i].process_time +
                                (sequence[i].arrival_time < completion_times1[i-1] ?
                                 completion_times1[i-1] : sequence[i].arrival_time);
      }
      stability_values[counter] = 0;
      for (i = 0; i < nremainingjobs; i++)
         stability_values[counter] += (completion_times[i] < completion_times1[i] ?
                    completion_times1[i] - completion_times[i] :
                    completion_times[i] - completion_times1[i] );
      for (i = 0; i < nremainingjobs; i++)
         sum += ((tardy = completion_times[i] - sequence[i].due_date) > 0 ? tardy : 0);
      tardiness_values[counter] = sum;
   }
   stabil = 0;
   sum = 0;
   for (counter = 0; counter < 5; counter++)
   {
      stabil += stability_values[counter];
      sum += tardiness_values[counter];
   }
   free(completion_times);
   free(completion_times1);
   stabil = stabil / 5.0;
   return (sum / 5.0);
}

double simulate_flowtime(job *sequence)
{
   double stability_values[5];
   double flowtime_values[5];
   double *completion_times;
   double *completion_times1;
   int i, counter;
   double tnow;
   int job_completed;
   double busy_time;
   double remaining_process;
   double sum;
   double next_failure;

   completion_times = (double *) malloc(nremainingjobs * sizeof(double));
```

```
completion_times1 = (double *) malloc(nremainingjobs * sizeof(double));
for (counter = 0; counter < 5; counter++)
{
   busy_time = 0;
   sum = 0;
   next_failure = get_failure();
   tnow = sequence[0].arrival_time;
   for (i = 0; i < nremainingjobs; i++)
   {
      job_completed = 0;
      remaining_process = sequence[i].process_time;
      if (busy_time == next_failure)
      {
         tnow += get_repair();
         busy_time = 0;
         next_failure = get_failure();
      }
      if (tnow < sequence[i].arrival_time) tnow = sequence[i].arrival_time;
      do
      {
         if (remaining_process > next_failure - busy_time)
         {
             tnow += (get_repair() + next_failure - busy_time);
             remaining_process -= (next_failure - busy_time);
             busy_time = 0;
             next_failure = get_failure();
         }
         else
         {
            busy_time += remaining_process;
            tnow += remaining_process;
            job_completed = 1;
         }
      }
      while (!job_completed);
      completion_times[i] = tnow;
   }
   completion_times1[0] = sequence[0].process_time + sequence[0].arrival_time;
   for (i = 1; i < nremainingjobs; i++)
   {
      completion_times1[i] = sequence[i].process_time +
                             (sequence[i].arrival_time < completion_times1[i-1] ?
                              completion_times1[i-1] : sequence[i].arrival_time);
   }
   stability_values[counter] = 0;
   for (i = 0; i < nremainingjobs; i++)
      stability_values[counter] += (completion_times[i] < completion_times1[i] ?
                   completion_times1[i] - completion_times[i] :
                   completion_times[i] - completion_times1[i] );
   for (i = 0; i < nremainingjobs; i++)
      sum += (completion_times[i] - sequence[i].arrival_time);
   flowtime_values[counter] = sum;
}
sum = 0;
```

```
   stabil = 0;
   for (counter = 0; counter < 5; counter++)
   {
      stabil += stability_values[counter];
      sum += flowtime_values[counter];
   }
   free(completion_times);
   free(completion_times1);
   stabil = stabil / 5.0;
   return (sum / 5.0);
}

int is_tabu(int job, int position)
{
   tabu *tp;
   int found = 0;

   for (tp = tabu_list; tp != NULL; tp = tp->next)
      if (tp->job_id == job && tp->to_position == position)
      {
         found = 1;
         break;
      }
   return found;
}

void generate_neighbourhood(void)
{
   int i, j, k = 0;
   job temp, *sequence;

   sequence = (job *) malloc(nremainingjobs * sizeof(job));
   for (i = 0; i < nremainingjobs; i++)
      for (j = i + 1; j < nremainingjobs; j++)
      {
         memcpy(sequence, current_solution, nremainingjobs * sizeof(job));
         neighbourhood[k].moves.first_job_id = sequence[i].id;
         neighbourhood[k].moves.first_job_from_position = i;
         neighbourhood[k].moves.first_job_to_position = j;
         neighbourhood[k].moves.second_job_id = sequence[j].id;
         neighbourhood[k].moves.second_job_from_position = j;
         neighbourhood[k].moves.second_job_to_position = i;
         temp = sequence[i];
         sequence[i] = sequence[j];
         sequence[j] = temp;
         memcpy(neighbourhood[k].sequence, sequence, nremainingjobs * sizeof(job));
         neighbourhood[k].obj_value = (*obj_array[objective])(sequence, weight);
         k++;
      }
   free(sequence);
}
```

```
int compar(element *i, element *j)
{
   return i->obj_value < j->obj_value ? -1 : i->obj_value > j->obj_value ? 1 : 0;
}

int compar_id(job *i, job *j)
{
   return i->id < j->id ? -1 : i->id > j->id ? 1 : 0;
}

int compar_arr(job *i, job *j)
{
   return i->arrival_time < j->arrival_time ? -1 : i->arrival_time > j->arrival_time ?
: 0;
}

int compar_edd(job *i, job *j)
{
   return i->due_date < j->due_date ? -1 : i->due_date > j->due_date ? 1 : 0;
}

int compar_spt(job *i, job *j)
{
   return i->process_time < j->process_time ? -1 : i->process_time > j->process_time ?
: 0;
}

void update(void)
{
   tabu *tp, *ip;

   for (tp = tabu_list; tp != NULL; tp = tp->next)
      --(tp->remaining_tenure);
   tp = tabu_list;
   while (tp != NULL)
   {
      if(tp->remaining_tenure < 1)
      {
         if (tp == tabu_list)
         {
            tabu_list = tp->next;
            free(tp);
         }
         else if (tp == tail)
         {
            for(ip = tabu_list; ip != NULL; ip = ip->next)
               if(ip->next == tp)
               {
                  tail = ip;
                  free(tp);
               }
         }
```

```
         else
         {
            for(ip = tabu_list; ip != NULL; ip = ip->next)
               if(ip->next == tp)
               {
                  ip->next = tp->next;
                  free(tp);
               }
         }
         tp = tabu_list;
      }
      else tp = tp->next;
   }
}

void process_neighbourhood(void)
{
   int i = 0;
   int found = 0;

   qsort(neighbourhood, nremainingjobs * (nremainingjobs -1 ) / 2, sizeof(element), (vc
*) compar);
   if (verbose) fprintf(g, "\nSearching current neighbourhood...\n");
   if (tabu_list == NULL)
   {

      if (verbose)
      {
         fprintf(g, "Tabu list is empty now. Jumping to the best neighbour.\n");
         fprintf(g, "\tSwapping job %d with job %d\n",
                    neighbourhood[0].moves.first_job_id,
                    neighbourhood[0].moves.second_job_id);
      }
      tabu_list = (tabu *) malloc (sizeof(tabu));
      tail = tabu_list;
      tail->job_id = neighbourhood[0].moves.first_job_id;
      tail->to_position = neighbourhood[0].moves.first_job_from_position;
      tail->remaining_tenure = tenure;
      tail->next = (tabu *) malloc(sizeof(tabu));
      tail = tail->next;
      tail->job_id = neighbourhood[0].moves.second_job_id;
      tail->to_position = neighbourhood[0].moves.second_job_from_position;
      tail->remaining_tenure = tenure;
      tail->next = NULL;
      memcpy(current_solution, neighbourhood[0].sequence, nremainingjobs * sizeof(job))
      current_obj = neighbourhood[0].obj_value;
      if (neighbourhood[0].obj_value < best_obj)
      {
         if (improve)
            fprintf(g, "Iteration: %d, New best solution with objective %f\n"
                     , iIter + 1, neighbourhood[0].obj_value);
         if (iIter - last_improved > interval) interval = iIter - last_improved;
         last_improved = iIter;
         counter = 0;
```

```
            memcpy(best_solution, current_solution, nremainingjobs * sizeof(job));
            best_obj = neighbourhood[0].obj_value;
         }
         return;
      }
      else
      {
         for(i=0; i < nremainingjobs * (nremainingjobs - 1) / 2; i++)
         {
            if(is_tabu(neighbourhood[i].moves.first_job_id,
                        neighbourhood[i].moves.first_job_to_position)
               || is_tabu(neighbourhood[i].moves.second_job_id,
                  neighbourhood[i].moves.second_job_to_position))
            {
               if(neighbourhood[i].obj_value < best_obj)
               {
                  if (verbose)
                  {
                     found = 1;
                     fprintf(g, "\tSwap job %d with job %d\t(Tabu but better than the best
                                 "schedule found upto now)\n",
                           neighbourhood[i].moves.first_job_id,
                           neighbourhood[i].moves.second_job_id);
                  }
                  break;
               }
               else
               {
                  if (verbose)
                  {
                     fprintf(g, "\tSwap job %d with job %d\t(Tabu, skipping it)\n",
                           neighbourhood[i].moves.first_job_id,
                           neighbourhood[i].moves.second_job_id);
                  }
                  continue;
               }
            }
            else
            {
               found = 1;
               if (verbose)
               {
                  fprintf(g,"\tSwap job %d with job %d\t(OK...)\n",
                        neighbourhood[i].moves.first_job_id,
                        neighbourhood[i].moves.second_job_id);
               }
               break;
            }
         }
         if(found)
         {
            if(!is_tabu(neighbourhood[i].moves.first_job_id,
               neighbourhood[i].moves.first_job_from_position))
```

```
         {
            tail->next = (tabu *) malloc (sizeof(tabu));
            tail = tail->next;
            tail->job_id = neighbourhood[i].moves.first_job_id;
            tail->to_position = neighbourhood[i].moves.first_job_from_position;
            tail->remaining_tenure = tenure + 1;
            tail->next = NULL;
         }
         if(!is_tabu(neighbourhood[i].moves.second_job_id,
            neighbourhood[i].moves.second_job_from_position))
         {
            tail->next = (tabu *) malloc (sizeof(tabu));
            tail = tail->next;
            tail->job_id = neighbourhood[i].moves.second_job_id;
            tail->to_position = neighbourhood[i].moves.second_job_from_position;
            tail->remaining_tenure = tenure + 1;
            tail->next = NULL;
         }
         memcpy(current_solution, neighbourhood[i].sequence, nremainingjobs *
sizeof(job));
         current_obj = neighbourhood[i].obj_value;
         if (neighbourhood[i].obj_value < best_obj)
         {
            if (improve)
               fprintf(g, "Iteration: %d, New best solution with objective %f\n"
                        , iIter + 1, neighbourhood[0].obj_value);
            if (iIter - last_improved > interval) interval = iIter - last_improved;
            last_improved = iIter;
            counter = 0;
            memcpy(best_solution, current_solution, nremainingjobs * sizeof(job));
            best_obj = neighbourhood[i].obj_value;
         }
      }
      else
      {
         if (verbose)
         {
            fprintf(g,"All neighbouring solutions move involves tabu moves so"
                   " jumping to the best neighbouring solution.\n");
            fprintf(g,"\tSwap job %d with job %d\n",
                    neighbourhood[0].moves.first_job_id,
                    neighbourhood[0].moves.second_job_id);
         }
         memcpy(current_solution, neighbourhood[0].sequence, nremainingjobs *
sizeof(job));
         current_obj = neighbourhood[0].obj_value;
         if (neighbourhood[0].obj_value < best_obj)
         {
            if (improve)
               fprintf(g, "Iteration: %d, New best solution with objective %f\n"
                        , iIter + 1, neighbourhood[0].obj_value);
            if (iIter - last_improved > interval) interval = iIter - last_improved;
            last_improved = iIter;
            counter = 0;
```

```
            memcpy(best_solution, current_solution, nremainingjobs * sizeof(job));
            best_obj = neighbourhood[0].obj_value;
         }
      }
      update();
   }
}

void print_usage (FILE *stream, int exit_code)
{
   fprintf(stream, "Usage: %s [-hivc] [-o <filename>] <input file>\n", program_name);
   fprintf(stream, "where available options are:\n"
           "  -h              Display this usage information\n"
           "  -o  <filename>  Write output to file\n"
           "  -i              Prints a message when current best is improved\n"
           "  -v              Print verbose messages.\n"
           "  -c              Continuous scheduling (after every m/c breakdown).\n");
   exit(exit_code);
}

job *ts(job *sequence)
{
   job *arr;
   job *edd;
   job *spt;
   tabu *tp;
   int i;

   arr = (job *) malloc(nremainingjobs * sizeof(job));
   edd = (job *) malloc(nremainingjobs * sizeof(job));
   spt = (job *) malloc(nremainingjobs * sizeof(job));
   memcpy(arr, sequence, nremainingjobs * sizeof(job));
   memcpy(edd, sequence, nremainingjobs * sizeof(job));
   memcpy(spt, sequence, nremainingjobs * sizeof(job));
   qsort(arr, nremainingjobs, sizeof(job), (void *) compar_arr);
   qsort(edd, nremainingjobs, sizeof(job), (void *) compar_edd);
   qsort(spt, nremainingjobs, sizeof(job), (void *) compar_spt);
   current_solution = (job *) malloc(nremainingjobs * sizeof(job));
   memcpy(current_solution, sequence, nremainingjobs * sizeof(job));
   current_obj = (*obj_array[objective])(current_solution, weight);
   memcpy(best_solution, sequence, nremainingjobs * sizeof(job));
   best_obj = current_obj;
   neighbourhood = (element *) malloc(nremainingjobs * (nremainingjobs - 1) *
sizeof(element) / 2);
   for (i = 0; i < (nremainingjobs - 1) * nremainingjobs / 2; i++)
      neighbourhood[i].sequence = (job *) malloc(nremainingjobs * sizeof(job));
   iIter = 0;
   for (counter = 0;; iIter++, counter++)
   {
      if (verbose)
      {
         fprintf(g, "ITERATION %d\n", iIter+1);
         fprintf(g, "\nCurrent sequence: ");
         for (i = 0; i < nremainingjobs; i++)
```

```
            fprintf(g, "%d ", current_solution[i].id);
         fprintf(g, "\tObjective: %f.\n", current_obj);
      }
      generate_neighbourhood();
      process_neighbourhood();
      if (verbose)
      {
         fprintf(g, "\nCurrent tabu list:\n");
         for(tp = tabu_list; tp != NULL; tp = tp->next)
            fprintf(g, "\tJob %d to position %d for %d %s.\n",
                   tp->job_id, tp->to_position + 1, tp->remaining_tenure,
                   tp->remaining_tenure > 1 ? "iterations" : "iteration");
         fprintf(g, "\n");
      }
      if (counter >= 20) break;
   }

   memcpy(current_solution, arr, nremainingjobs * sizeof(job));
   current_obj = (*obj_array[objective])(current_solution, weight);
   for (counter = 0;; iIter++, counter++)
   {
      if (verbose)
      {
         fprintf(g, "ITERATION %d\n", iIter+1);
         fprintf(g, "\nCurrent sequence: ");
         for (i = 0; i < nremainingjobs; i++)
            fprintf(g, "%d ", current_solution[i].id);
         fprintf(g, "\tObjective: %f.\n", current_obj);
      }
      generate_neighbourhood();
      process_neighbourhood();
      if (verbose)
      {
         fprintf(g, "\nCurrent tabu list:\n");
         for(tp = tabu_list; tp != NULL; tp = tp->next)
            fprintf(g, "\tJob %d to position %d for %d %s.\n",
                   tp->job_id, tp->to_position + 1, tp->remaining_tenure,
                   tp->remaining_tenure > 1 ? "iterations" : "iteration");
         fprintf(g, "\n");
      }
      if (counter >= 20) break;
   }
   memcpy(current_solution, edd, nremainingjobs * sizeof(job));
   current_obj = (*obj_array[objective])(current_solution, weight);
   for (counter = 0;; iIter++, counter++)
   {
      if (verbose)
      {
         fprintf(g, "ITERATION %d\n", iIter+1);
         fprintf(g, "\nCurrent sequence: ");
         for (i = 0; i < nremainingjobs; i++)
            fprintf(g, "%d ", current_solution[i].id);
         fprintf(g, "\tObjective: %f.\n", current_obj);
      }
```

```
      generate_neighbourhood();
      process_neighbourhood();
      if (verbose)
      {
         fprintf(g, "\nCurrent tabu list:\n");
         for(tp = tabu_list; tp != NULL; tp = tp->next)
            fprintf(g, "\tJob %d to position %d for %d %s.\n",
                   tp->job_id, tp->to_position + 1, tp->remaining_tenure,
                   tp->remaining_tenure > 1 ? "iterations" : "iteration");
         fprintf(g, "\n");
      }
      if (counter >= 20) break;
   }
   memcpy(current_solution, spt, nremainingjobs * sizeof(job));
   current_obj = (*obj_array[objective])(current_solution, weight);
   for (counter = 0;; iIter++, counter++)
   {
      if (verbose)
      {
         fprintf(g, "ITERATION %d\n", iIter+1);
         fprintf(g, "\nCurrent sequence: ");
         for (i = 0; i < nremainingjobs; i++)
            fprintf(g, "%d ", current_solution[i].id);
         fprintf(g, "\tObjective: %f.\n", current_obj);
      }
      generate_neighbourhood();
      process_neighbourhood();
      if (verbose)
      {
         fprintf(g, "\nCurrent tabu list:\n");
         for(tp = tabu_list; tp != NULL; tp = tp->next)
            fprintf(g, "\tJob %d to position %d for %d %s.\n",
                   tp->job_id, tp->to_position + 1, tp->remaining_tenure,
                   tp->remaining_tenure > 1 ? "iterations" : "iteration");
         fprintf(g, "\n");
      }
      if (counter >= 20) break;
   }
   for (tp = tabu_list; tp != NULL; tp = tp->next)
      tp->remaining_tenure = 1;
   update();
   free(current_solution);
   free(arr);
   free(edd);
   free(spt);
   for (i = 0; i < (nremainingjobs - 1) * nremainingjobs / 2; i++)
      free(neighbourhood[i].sequence);
   free(neighbourhood);
   return (best_solution);
}
```

```
int main(int argc, char *argv[])
{
   double yedek;

   int c;
   int totalproc = 0;
   extern char *optarg;
   extern int optind;
   const char *output_filename = NULL;
   schedule *initial_schedule;
   schedule *realized_schedule;
   job *alias, *head;
   job *temp_sequence;
   int i;
   double sum, tardy;
   int ntotaljobs, next_one;
   double tnow, busy_time, next_failure;

   srand(time(NULL));
   program_name = argv[0];
   while ((c = getopt(argc, argv, "hivco:")) != EOF)
      switch (c)
      {
         case 'h': print_usage(stdout, 0);
         case 'i': improve = 1; break;
         case 'v': verbose = 1; break;
         case 'c': continuous = 1; break;
         case 'o': output_filename = optarg; break;
         case '?': print_usage(stderr, 1);
      }

   if (argv[optind] == NULL) print_usage(stderr, 1);

   if ((f = fopen(argv[optind], "rt")) == NULL)
   {
      char buffer[80];
      sprintf(buffer, "Cannot open %s", argv[optind]);
      perror(buffer);
      exit(1);
   }

   if (output_filename == NULL) g = stdout;
   else if ((g = fopen(output_filename, "wt")) == NULL)
   {
      char buffer[80];
      sprintf(buffer, "Cannot open %s", output_filename);
      perror(buffer);
      exit(1);
   }

   /* memory initializations */

   obj_array[0] = makespan;
   obj_array[1] = tardiness;
```

```
obj_array[2] = flowtime;
obj_array[3] = slack_makespan;
obj_array[4] = slack_tardiness;
obj_array[5] = slack_flowtime;
obj_array[6] = method1_makespan;
obj_array[7] = method1_tardiness;
obj_array[8] = method1_flowtime;
obj_array[9] = method2_makespan;
obj_array[10] = method2_tardiness;
obj_array[11] = method2_flowtime;
obj_array[12] = method1_stability;
obj_array[13] = method2_stability;

nremainingjobs = fread_number(f);
ntotaljobs = nremainingjobs;
jobs = (job *) malloc(ntotaljobs * sizeof(job));
for (i = 0; i < ntotaljobs; jobs[i].id = ++i)
   jobs[i].process_time = fread_number(f);
for (i = 0; i < ntotaljobs; jobs[i].id = ++i)
   jobs[i].arrival_time = fread_number(f);
for (i = 0; i < ntotaljobs; i++)
   jobs[i].due_date = fread_number(f);
objective = fread_number(f);
performance_measure = fread_number(f);
weight = fread_number(f);
tenure = fread_number(f);
breakdown = fread_number(f);
fclose(f);
switch(breakdown)
{
   case 0: mean_time_to_failure = 60.0;
           mean_repair_duration = 9.0;
           break;
   case 1: mean_time_to_failure = 60.0;
           mean_repair_duration = 3.0;
           break;
   case 2: mean_time_to_failure = 18.0;
           mean_repair_duration = 9.0;
           break;
   case 3: mean_time_to_failure = 18.0;
           mean_repair_duration = 3.0;
           break;
}
lambda = 0.7 / mean_repair_duration;
for (i = 0; i < ntotaljobs; i++) totalproc += jobs[i].process_time;
cdf = (double *) malloc((totalproc + 1) * sizeof(double));
best_solution = (job *) malloc(ntotaljobs * sizeof(job));
if (continuous && (objective > 8))
{
   cdf[0] = 0.0;
   for (i = 1; i <= totalproc; i++)
      cdf[i] = cdf[i - 1] + integrate(failure_dist, i - 1, i);
}
```

```
   /* main work */

   alias = ts(jobs);

   if (!continuous)
   {
      fprintf(g, "%s\t%f\t%f\t", argv[optind],
                  (*obj_array[performance_measure])(best_solution, weight)
                  ,((performance_measure == 0 )? simulate_makespan(best_solution)
                  :(performance_measure == 1 )? simulate_tardiness(best_solution)
                  : simulate_flowtime(best_solution)));
      fprintf(g, "%f\n", stabil);
      free(jobs);
      free(best_solution);
      free(cdf);
      return 0;
   }
   yedek = (*obj_array[performance_measure])(best_solution, weight);
   initial_schedule = (schedule *) malloc(ntotaljobs * sizeof(schedule));
   initial_schedule[0].id = best_solution[0].id;
   initial_schedule[0].completion_time = best_solution[0].arrival_time +
best_solution[0].process_time;
   for (i = 1; i < ntotaljobs; i++)
   {
      initial_schedule[i].id = best_solution[i].id;
      initial_schedule[i].completion_time = best_solution[i].process_time +
        ((initial_schedule[i-1].completion_time < best_solution[i].arrival_time) ?
         best_solution[i].arrival_time : initial_schedule[i-1].completion_time);
   }
   realized_schedule = (schedule *) malloc(ntotaljobs * sizeof(schedule));
   tnow = best_solution[0].arrival_time;
   busy_time = 0;
   next_one = 0;
   while (nremainingjobs > 0)
   {
      if (tnow < alias->arrival_time) tnow = alias->arrival_time;
      next_failure = get_failure();
      while (nremainingjobs > 0)
      {
         if (tnow < alias->arrival_time) tnow = alias->arrival_time;
         if (busy_time + alias->process_time < next_failure)
         {
            busy_time += alias->process_time;
            tnow += alias->process_time;
            nremainingjobs--;
            realized_schedule[next_one].id = alias->id;
            realized_schedule[next_one++].completion_time = tnow;
            alias++;
         }
         else if (busy_time + alias->process_time > next_failure)
         {
            tnow += (next_failure - busy_time + get_repair());
            (alias->process_time) -= (next_failure - busy_time);
```

```
            busy_time = 0;
            head = alias;
            for (i = 0; i < nremainingjobs; i++)
            {
                alias->arrival_time = (alias->arrival_time < tnow ? 0 : (int)
(alias->arrival_time - tnow));
                alias->due_date = (int)(alias->due_date - tnow);
                alias++;
            }
            temp_sequence = (job *) malloc (nremainingjobs * sizeof(job));
            memcpy(temp_sequence, head, nremainingjobs * sizeof(job));
            free(best_solution);
            best_solution = (job *) malloc(nremainingjobs * sizeof(job));
            if (nremainingjobs > 1) alias = ts(temp_sequence);
            else
            {
               alias = best_solution;
               memcpy(best_solution, temp_sequence, sizeof(job));
            }
            for (i = 0; i < nremainingjobs; i++)
            {
                best_solution[i].arrival_time = jobs[best_solution[i].id-1].arrival_tim
                best_solution[i].due_date = jobs[best_solution[i].id-1].due_date;
            }
            free(temp_sequence);
            break;
         }
         else
         {
            tnow += alias->process_time;
            busy_time = 0;
            nremainingjobs--;
            realized_schedule[next_one].id = alias->id;
            realized_schedule[next_one++].completion_time = tnow;
            alias++;
            head = alias;
            for (i =0; i < nremainingjobs; i++)
            {
                alias->arrival_time = alias->arrival_time < tnow ? 0 : (int) (alias-
>arrival_time - tnow);
                alias->due_date = (int)(alias->due_date - tnow);
                alias++;
            }
            temp_sequence = (job *) malloc (nremainingjobs * sizeof(job));
            memcpy(temp_sequence, head, nremainingjobs * sizeof(job));
            free(best_solution);
            best_solution = (job *) malloc(nremainingjobs * sizeof(job));
            if (nremainingjobs > 1) alias = ts(temp_sequence);
            else
            {
               alias = best_solution;
               memcpy(best_solution, temp_sequence, sizeof(job));
            }
            free(temp_sequence);
```

```
            for (i = 0; i < nremainingjobs; i++)
            {
                best_solution[i].arrival_time = jobs[best_solution[i].id-1].arrival_tim
                best_solution[i].due_date = jobs[best_solution[i].id-1].due_date;
            }
            break;
         }
      }
   }

   sum = 0;
   switch(performance_measure)
   {
      case 0 : sum = realized_schedule[ntotaljobs - 1].completion_time;
               break;
      case 1 : for (i = 0; i < ntotaljobs; i++)
                  sum += ((tardy = realized_schedule[i].completion_time
                          - jobs[realized_schedule[i].id-1].due_date) > 0 ? tardy : 0)
               break;
      case 2 : for (i = 0; i < ntotaljobs; i++)
                  sum += (realized_schedule[i].completion_time -
jobs[realized_schedule[i].id-1].arrival_time);
               break;
   }
   stabil = 0;
   qsort(initial_schedule, ntotaljobs, sizeof(schedule), (void *) compar_id);
   qsort(realized_schedule, ntotaljobs, sizeof(schedule), (void *) compar_id);
   for (i = 0; i < ntotaljobs; i++)
      stabil += (realized_schedule[i].completion_time >
initial_schedule[i].completion_time ?
                 (realized_schedule[i].completion_time -
initial_schedule[i].completion_time) :
                 (initial_schedule[i].completion_time -
realized_schedule[i].completion_time));
   fprintf(g, "%s\t%f\t%f\t%f\n", argv[optind], yedek, sum, stabil);

   /* memory clean-up before exit */

   free(jobs);
   free(best_solution);
   free(cdf);
   free(initial_schedule);
   return 0;
}
```