Evaluation of On-Line Scheduling Rules for High Volume Job Shop Problems, a Simulation Study

Yoni Nazarathy

THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE M.A. DEGREE

University of Haifa
Faculty of Social Sciences
Department of Statistics

September, 2001

Evaluation of On-Line Scheduling Rules for High Volume Job Shop Problems, a Simulation Study

By: Yoni Nazarathy

Supervised by: Professor Gideon Weiss

THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE M.A. DEGREE

University of Haifa
Faculty of Social Sciences
Department of Statistics

September, 2001

Approved by: ______________________________ Date: _______
(Supervisor)

Approved by: ______________________________ Date: _______
(Chairman of M.A Committee)

Thank you page goes here

Contents

Evaluation of On-Line Scheduling Rules for High Volume Job Shop Problems, a Simulation Study.

Yoni Nazarathy

Abstract

We discuss scheduling of *high volume* job shop problems with respect to makespan using simple on-line scheduling rules. We say that a job shop problem is high volume if the number of jobs is large relative to the number of machines and the maximal route length. We focus on an algorithm which imitates the optimal fluid solution of this problem, which we name the Makespan Fluid Imitation Algorithm ($C_{\max}$ FIA). To learn more about $C_{\max}$ FIA, we designed the Job Shop Simulation Project, a software package that is designed to simulate high volume job shop problems that are driven by on-line scheduling heuristics.

Our simulation results show that the makespan that is achieved by the $C_{\max}$ FIA heuristic is equal to the machine lower bound plus a constant that is independent of the number of jobs. This is true for a variety of processing time distributions, including heavy tailed distributions. For job shop problems with several bottleneck machines, our results are the same, given that for each bottleneck machine, there is a route which starts at that machine. In addition, our results hint that a simple heuristic that operates by selecting buffers at random yields a makespan that is asymptotically optimal. While the performance of this random scheduling scheme is not as good as the $C_{\max}$ FIA, it is now evident that asymptotically optimal scheduling for the high volume job shop problem is easily achievable.

We summarize our results in a set of conjectures that are based on the extensive simulation study. These conjectures are yet to be proved. In addition we discuss the relationship between the stability of MCQNs and asymptotically optimal job shop scheduling in terms of makespan. Attempting to investigate a simple yet interesting special case, we present a steady-state model of a job shop with 2 machines, and 2 routes which run in opposite directions. While we do not solve it, we believe that this model has a steady state solution. Our belief is based on the simulation results.

List of Tables

List of Figures

List of Symbols and Abbreviations

The Job Shop Problem

The number of jobs	J
The number of machines	I
The number of routes	R
The number of operations in route r	K_r
The number of jobs on route r	N_r
The o'th operation in route r	(r,o)
The machine on which operation (r, o) is processed	$\sigma(r, o)$
The route of job j	$\rho(j)$
The operations performed on machine i (the constituency of machine i)	C_i
The processing time of job j on operation (r, o)	$X_{(r,o)}(j)$
The mean processing time of operation (r, o)	$m_{(r,o)}$
The bottleneck machine	i^*

Schedules and Objective Functions

A schedule	$\mathcal{S}$
Start time of operation (r, o) for job j	$s_{(r,o)}(j)$
The completion time of operation (r, o) for job j	$t_{(r,o)}(j)$
The completion time of job j	c_j
The amount of jobs at buffer (r, o) at time t	$Q_{(r,o)}(t)$
The amount of upstream jobs for buffer (r, o) at time t	$Q^+_{(r,o)}(t)$
The makespan	$C_{\max}$
Work-in-process inventory costs (also known as weighted flow time)	WIT

The machine lower bound	T^*
The $C_{\max}$ generated by a heuristic	T^H
The $C_{\max}$ generated by the optimal schedule	T^{opt}
The gap: $T^H - T^*$	$\mathcal{G}$
The starve time	$\Im$
The run-out time	$\Re$
The voluntary rest time	$\mho$

The High Volume Job Shop Problem

A high volume job shop problem with a fixed number of routes	$R \ll J$
A high volume job shop problem with arbitrary routes	$R \approx J$
The multiplicity	N
The log-multiplicity	$\log N$

Scheduling Algorithms / Heuristics / Rules

A scheduling algorithm	Υ
The optimal scheduling algorithm	Υ^{opt}
Fluid imitation algorithm for makespan	$C_{\max}$ FIA
Random buffer scheduling rule	RBSR
Proportional random buffer scheduling rule	PRBSR
Random job scheduling rule	RJSR
First buffer first serve	FBFS
Last buffer first serve	LBFS

Fluid Models

The amount of fluid in buffer (r, o) at time t	$q_{(r,o)}(t)$
The amount of upstream fluid for buffer (r, o) at time t	$q^+_{(r,o)}(t)$
The rate of flow out of buffer (r, o) at time t	$u_{(r,o)}(t)$
The lag of buffer k as used by the FIA	L_k

Simulation Experiments of the Job Shop Problem

A job shop problem instance	$\wp$
The population of all job shop problems	$\mathcal{P}$
A sub-population of job shop problems (for given class i)	$\mathcal{P}_i$
Job shop problem instances of multiplicity N from $\mathcal{P}_i$	$\mathcal{P}_{i,N}$
A schedule generated by applying Υ to $\wp$	$\mathcal{S}_{\Upsilon}(\wp)$
The number of replications	n

Analysis of LBFS Non-idling policy on a Re-entrant Line (Section 5.2)

The number of iterations of the calculating algorithm	L
The buffer that is set as the initial bottleneck buffer in the l'th iteration	$a^{(l)}$
The number of jobs that are finished on k after working for a duration of t	$S_k(t)$
The amount of time that machine $\sigma(k)$ works on buffer k during $[0, t)$	$T_k(t)$

Probability and Stochastic Processes

Random variable	R.V.
Coefficient of variation (standard deviance divided by the mean)	C.V.
Independent and identically distributed random variables	i.i.d.
Central limit theorem	CLT
Moment generating function	MGF
Multi class queueing networks	MCQN

Software Design, JAVA and the JSSP

Object oriented	OO
OO design	OOD
Graphical user interface	GUI
JAVA virtual machine	JVM
Job Shop Simulation Project	JSSP
Job Shop Simulator	JSS
Simulation results database	SRDB

Overview

This reports summarizes a simulation study that investigates solution methods of the high volume job shop problem (a job shop problem with a large number of jobs). We categorize instances of the problem into two main categories: The $R \ll J$ case and the $R \approx J$ case. In the former, the number of routes (R) is much smaller than the number of jobs (J). In the later, each job may follow any route.

The main goal of this study was to investigate how well simple on-line scheduling heuristics are able to schedule the $R \ll J$ high volume job shop with respect to makespan ($C_{\max}$). Recent results by Bertsimas and Gamarnik [4], Dai and Weiss [14] and Boudoukh, Penn and Weiss ([6] and [7]) have shown that it is possible to obtain schedules that obtain a $C_{\max}$ that is close to the optimal schedule. These results are based on the machine lower bound (T^*), achieving makespans that are equal to $T^* + o(N)$. We call such schedules asymptotically optimal due to the fact that the ratio $(C_{\max} - T^*)/T^*$ approaches 0 as the number of jobs increases. Note that some of these results (Boudoukh et. al. and Bertsimas and Gamarnik) are minimax results while others (Dai and Weiss) are probabilistic results.

We performed an extensive computer simulation of job shop problem instances using the following on-line scheduling heuristics: random job scheduling rule (RJSR), random buffer scheduling rule (RBSR), proportional random buffer scheduling rule (PRBSR), first buffer first serve (FBFS), last buffer first server (LBFS) and the fluid imitation algorithm for makespan ($C_{\max}$ FIA). It was the performance of the last heuristic ($C_{\max}$ FIA) that we were extremely interested in investigating. For the purpose of the simulation, we created the Job Shop Simulation Project (JSSP), a software package that allowed for easy and efficient execution of $R \ll J$ job shop simulations.

The RJSR, RBSR and PRBSR are simple to execute, randomly behaving scheduling rules. We describe these within the body of the text. The FBFS and LBFS scheduling rules are also simple to execute, deterministic scheduling rules. We examined these rules for the sole purpose of showing that some scheduling rules are not asymptotically optimal. The $C_{\max}$ FIA scheduling rule was presented as the greedy fluid algorithm (GFA) in [6] and [7]. This scheduling rule is based on the fluid solution to the fluid relaxation of the job shop problem, to be presented in the text. It attempts to schedule the job shop such that it emulates an optimal fluid solution.

We define the gap ($\mathcal{G}$) as the makespan obtained by the heuristic minus the machine lower bound ($\mathcal{G} = C_{\max} - T^*$). Our study is based on the behavior of the gap relative to the number of jobs. We simulate a variety of problems using the heuristics presented above and empirically measure the growth of the gap. Our analysis is an empirical probabilistic analysis. This means that we use empirical data (from simulations) to make probabilistic statements for which we currently do not have any proof. We are primarily interested in answering the following questions:

- How well does a fluid imitation algorithm (FIA) for minimization of makespan perform? The simulation conducted by Boudoukh [6] has shown that when there is an equal number of jobs on each route and when the variability of the processing times is mild (exponential moments exist) then the algorithm generates schedules with a gap that is bounded by a constant (with respect to the problem size). Does this still hold when the processing times are generated from heavy tailed distributions?

- Does it hold when the number of jobs on each route is not identical?

- How well does the FIA function on problems that have multiple bottleneck machines? The positive trait of a bottleneck machine is that it may be utilized for one hundred percent of the time. The negative trait is that it may slow down the operation of the job shop. A job shop with several bottleneck machines is desired given that there is an assurance that these machines will work without starving. Can this be achieved?

- In what way does the application of on-line scheduling heuristics differ between job shop problems with several routes and job shop problems with a single route (re-entrant lines)? This is an interesting question because of recent results regarding stability and instability of re-entrant lines when modeled as MCQNs (see [3], [12], [13] and [33]).

- How well do random on-line scheduling algorithms perform? If it could be shown that extremely simple random scheduling rules do as good a job as more advanced scheduling rules, then the importance of the advanced scheduling rules should be questioned.

This report is organized as follows: We start off in Chapter 1 with an introduction to the job shop problem, the high volume job shop problem, methods of solution of the problem and real life applications of the problem. In Chapter 2 we focus on on-line scheduling algorithms that attempt to approximate the optimal solution of the $R \ll J$ high volume job shop problem. We introduce fluid models and discuss the theoretical effectiveness of such algorithms. The algorithms presented in this chapter are the ones that were tested by means of simulation. In chapter 3 we introduce the software package that was developed to allow for investigation of the high volume job shop problems: the Job Shop Simulation Project. We discuss both the design and implementation issues that were encountered during the development of the project. In Chapter 4, we lay out the framework for the simulation experiments that were performed. We discuss the theory behind the experiments, and enumerate all of the runs that were made. We also summarize the results of the experiments, specifying the observed rate of growth of the gap. In Chapter 5 we discuss the results of these experiments, and attempt to explain some of the observed phenomena. We discuss theoretical justifications and make conjectures based on the results. In Chapter 6 we continue our analysis, discussing the case where there are multiple bottleneck machines. Here, we analyze a simple special case: a 2 machine, 4 operation job shop problem with 2 opposite routes. We summarize results regarding simulation experiments that were run for this model. We also lay the foundation for a steady-state queueing network model that is based on this finite-horizon job shop model and the FIA scheduling rule. Finally in Chapter 7 we conclude our results and pinpoint the interesting questions that were raised during study and should be answered in the future.

Chapter 1

The Job Shop Problem

In this chapter we introduce the *job shop problem*, a well known problem in the fields of operations research, industrial engineering and computer science. In Section 1.1 we define the general job shop problem and in Section 1.2 we introduce the basic concepts relating to the problem. In Section 1.3 we introduce the high volume job shop problem, this is the type of job shop problem that we analyze in this study. In Section 1.4, we survey existing job shop scheduling techniques. This is a survey of algorithms and heuristics that are used to schedule the job shop problem without making use of the fact that it may be high volume. We finish off this chapter in Section 1.5 where we briefly discuss real world applications of the job shop problem[1].

1.1 The General Job Shop Problem

The job shop problem is a model of a generic factory that produces products. Each product is called a *job* because it traverses through the *machines* in the factory on its way to completion. There is a known fixed *route* for each job. A *route* is described by a sequence of *operations*. Each operation is characterized by the machine on which it is performed and the duration of time that it takes (sometimes mean duration). We allow for *re-entry* (sometimes known as recirculation). This means that each route may be composed of several operations on the same machine. We also allow for *immediate re-entry*, meaning that two consecutive operations on the same route may be processed on the same machine.

We define the *general job shop problem* as follows: J jobs are to be scheduled on machines $i = 1, \ldots, I$. Each job follows a route r, composed of operations $(r, 1), \ldots, (r, K_r)$.

[1] Note that the term *job shop problem* is widely used in the literature. This is opposed to the *general job shop problem* and the *high volume job shop problem* which are terms that we are introducing in this study and defining in this chapter.

The routes are numbered r=1,...,R and the jobs are numbered j=1,...,J. It may be that $R = J$ and in that case each job follows its own distinct route. It may also be that $R < J$ and in that case one or more jobs share a route. Note that in the former case, the problem does not require the concept of a route and the terms job and route are synonymous. We define $\rho(j)$ to be the route of job j. We define $\sigma(r, o)$ as the machine on which operation (r, o) is performed. We denote by C_i the constituency of machine i: $C_i = \{(r, o) : \sigma(r, o) = i\}$. We denote the processing time of job j on operation (r, o) by $X_{(r,o)}(j)$, thus the processing times of job j on all the operations that it requires: $(\rho(j)$,1$),\ldots,(\rho(j)$,$K_{\rho(j)})$ are denoted by $X_{(\rho(j),1)}(j), \ldots, X_{(\rho(j),K_{\rho(j)})}(j)$. We denote the average processing time of operation (r, o) by $m_{(r,o)}$ (this is an average over all of the processing times of jobs that require this operation). We label the amount of jobs at buffer (r, o) (waiting for operation (r, o) besides machine $\sigma(r, o)$ or actually performing operation (r, o)) at time t by $Q_{(r,o)}(t)$.

We name the above model general because it does not impose restrictions requiring an equal number of jobs per route or many jobs to follow the exact same route. It also does not require all of the jobs sharing a route to share the same processing times.

It should be noted that the general job shop is a relativity poor model of a real world factory. If one was interested to create a more accurate model of a real world manufacturing or processing situation, one may want to add some more features to the model.

These are some of the features that may enhance the general job shop model, making it a more reasonable approximation of a real-life factory:

Machine Breakdowns : Certain machines may not be operating at certain times of maintenance or repair.

Non-unique Routes : A job may have the option of traversing one of several different routes on its way to completion.

Set-up Times : Machines that are designated to perform several operations may have to spend time setting up between one job and another. The duration of the set up times may depend on both the job which has just finished and the job that is to be processed.

Transfer Times : The transfer of jobs between machines requires a given amount of time.

Availability Dates : Not all jobs are available for processing at the start of the execution of the job shop.

Due Dates : Jobs are to be completed by certain times. If not, a penalty may be incurred.

See [44] for an introductory discussion on the above and other features that may be found in scheduling problems. While the above features are required to create a model that truly mimics a real world factory, we focus on the general job shop model as presented above. This is due to the fact that there are still many open questions regarding the general job shop model. See Section 1.5 for a discussion of real world applications of the general job shop problem.

1.2 Feasible Schedules, Objective Functions and Lower Bounds

A *schedule* for the General Job Shop Problem uses all the data in the problem. It specifically determines the start and finish times $s_{(r,o)}(j)$, $t_{(r,o)}(j)$ of each operation as well as the job completion times, c_j (note that $c_j = t_{(\rho(j),K_{\rho(j)})}(j)$). These are subject to a one to one assignment of jobs to machines, order of steps, release times and processing times as summarized in the equations and inequalities below:

$$t_{(r,o)}(j) = s_{(r,o)}(j) + X_{(r,o)}(j) \tag{1.1}$$

$$s_{(r,o)}(j) \geq t_{(r,o-1)}(j) \tag{1.2}$$

$$I_{((r,o),j)}(t) \in \{0,1\} \tag{1.3}$$

$$I_{((r,o),j)}(t) = 1 \Longleftrightarrow s_{(r,o)}(j) \leq t < t_{(r,o)}(j) \tag{1.4}$$

$$I_{((r,o))}(t) = \sum_{j=1}^{J} I_{((r,o),j)}(t) \tag{1.5}$$

$$\sum_{(r,o)\in C_i} I_{(r,o)} \in \{0,1\} \tag{1.6}$$

Constraint 1.1 ensures that there is no preemption in the schedule. Constraint 1.2 ensures that a job is processed in the required order and at one operation at a time. The definitions and constraints 1.3, 1.4, 1.5 and 1.6 ensure that a machine processes at most one job at a time. Denoting a schedule as $\mathcal{S}$, it can be seen that given the above constraints, the minimal information that is needed to specify $\mathcal{S}$ is the set of $s_{(r,o)}(j)$ values.

Solving a job shop problem means finding a feasible schedule which optimizes some objective function. We examine two objective functions, both of which are to be minimized (such objective functions are referred to as *cost functions*).

> *Makespan (C_{max})* : $\max_j c_j$.
>
> This is the time at which the last job is completed. It is the most widely discussed objective function within the context of job shop scheduling. The makespan is important when the number of jobs is finite. This objective is closely related to maximum throughput in the sense that schedules that feature a low makespan usually tend to maximize the throughput rate.
>
> *Work-in-Process Inventory Costs (WIT)* : $\sum_j w_j c_j$.
>
> This is a weighted sum of the completion times of each job. The weights w_j are to be set based on the inventory costs incurred by having job j in process. This measure is also frequently referred to as weighted flow time.

See [45] and [44] for a description of the above objective functions along side several other objective functions. While we believe that in a manufacturing setting, the WIT objective is often more important than the $C_{\max}$ objective, the discussion in this paper and the experiments of this study are mostly related to $C_{\max}$. This is due to the fact that until recently, WIT optimizing job shop schedules were nearly impossible to obtain. In Section 2.2.2 we briefly discuss recent advances that may lead to near optimal scheduling for minimization of WIT.

Given a Job Shop Instance, and a cost function, our goal is to find a schedule that minimizes cost. Before attempting to tackle this difficult task, we first find lower bounds for the cost function. For the $C_{\max}$ cost function, two well known lower bounds exist:

> *The Machine Lower Bound (T^*)* : By looking at the problem data, we may find the machine i^* that has the most work to do. We call i^* the *bottleneck machine* (there may be several bottlenecks and in that case we use $i_1^*, i_2^*, \ldots$). We denote T^* the duration of the work of i^*. Thus:
> $T^* = \max_{i=1,\ldots,I} \sum_{(r,o)\in C_i} \sum_{j:\rho(j)=r} X_{(r,o)}(j)$.

The Job Lower Bound : This is the duration of processing of the slowest job: $\max_j \sum_{o=1}^{K_{\rho(j)}} X_{(\rho(j),o)}(j)$.

Note that we are not aware of any solid lower bounds for the WIT cost function[2].

1.2.1 Idle Time Decomposition

We call the duration of time during which a machine is not performing any operation (and the job shop has still not terminated) the *idle time* of the machine[3].

Any machine may be idle because of one of two main reasons: (1) It has reached a state in which its buffers are empty. (2) The scheduling algorithm has decided that it should perform a voluntary rest (even though its buffers are not empty).

We denote the duration of time during which a machine's buffers are empty as *vacant time*. We denote the duration of time that a machine performs voluntary rest as the *voluntary rest time*.

We can further decompose idle time by decomposing the vacant time. Notice that during the period in which a machine's buffers are empty one of two states may occur: (a) Not all of the operations have been performed on the machine. (b) All of the operations have been performed (the machine has finished). We call the duration of time during which state (a) occurs the *starve time*. We call the duration of time during which state (b) occurs the *run-out time*.

Thus the starve time, run-out time and voluntary rest time decompose (add up to) the idle time of a machine. As we perform the simulation experiments of this study, we record these times for each simulation run that is performed (see Chapter 4).

We define a *non-idling policy* as a scheduling policy that produces a schedule with a voluntary rest time equal to zero.

1.3 The High Volume Job Shop Problem

We say that a job shop problem is *high volume*, if the number of jobs exceeds the number of machines by a huge factor while the maximal route length remains bounded. There are two types of high volume job shop problems that we deal with:

$R \ll J$: This is the case in which the number of routes is much smaller than the number of jobs, implying that there are many jobs following the same

[2] In [5], Bertsimas et al. show that $F(N) - O(N)$ is a lower bound for the WIT cost function. Where $F(N)$ is a solution to a fluid model that is formulated.

[3] For a bottleneck machine, the idle time equals $C_{\max} - T^*$.

route. We allow the processing times of each of the steps of each of the J jobs to vary.

$R \approx J$: This is the case in which the number of routes is just about equal to the number of jobs, meaning that each job (or almost each job) follows a distinct route. In this case, even though $\max_r K_r$ and I are small with respect to J (as specified for the high volume job shop problem), the number of possible routes of length K is I^K. Thus even for moderately sized I and $\max_r K_r$, instances of the type $R \approx J$ are obtainable.

In the $R \ll J$ case, we denote by N_r the number of jobs on route r. We decompose the $R \ll J$ case into two sub cases, *identical* and *proportional*. We say a problem is identical if all N_r are equal. In that case we define $N_r = N$. We say the problem is proportional if it is not identical. In the proportional case we denote $N = \sum_{r=1}^{R} N_r$. Note that in the proportional case, $J = N$ whereas in the identical case $J = RN$.

We call N the *multiplicity* of the problem. In the $R \approx J$ case, $N = J = R$ denotes the multiplicity. We denote $\log N$ as the *log-multiplicity*. In the experiments we performed in this study, we increase the multiplicities exponentially thus we are increasing the log-multiplicities linearly. Note that throughout the study we use base 2 logarithms.

It is important to point out a special case of the $R \ll J$ problem: the *re-entrant line*. This is a high volume job shop problem with a single route ($R = 1$) and more operations than machines ($K_1 > I$). It has been studied within the context of multi class fluid networks, see [33], [12] and [13]. We will refer to it at occasions within the text.

1.3.1 Probabilistic Statements Regarding the High Volume Job Shop Problem and the Machine Lower Bound

All of the experiments that we perform and conjectures that we formulate are of a probabilistic type. When we make statements regarding the behavior of scheduling algorithms on job shop problems, we are actually referring to a population of job shop problems drawn from some distribution (see Chapter 4 for a formal description). For the $R \ll J$ case, we assume the existence of mean processing times $m_{(r,o)}$ for each operation (r, o). This assumption implies that as we increase the multiplicity, thus creating more jobs on each route, each job has a processing time $X_{(r,o)}(j)$ sampled from a distribution having the mean $m_{(r,o)}$. For the $R \approx J$ case, there is only one job performed on each operation (r, o). In this case we assume that the means m_i exist, these are the mean processing times of all jobs on machine i. As we increase the multiplicity, we assume that the means remain constant. This is a reasonable assumption in a manufacturing setting. Note that we also assume that the processing times have a finite variance (finite second moment).

It is important to understand the difference between a probabilistic analysis and a minimax analysis. In a a minimax analysis one tries to make statements regarding the worst case problem instance that may occur. In the probabilistic analysis, statements that are true with a high probability are made (a probability that usually approaches 1 as the multiplicity increases). For an example of a probabilistic study of the job Shop problem see [14].

We will be evaluating scheduling algorithms that attempt to approximate the optimal schedule of high volume job shop problems. We denote the $C_{\max}$ generated by an optimal schedule as T^{opt}. A schedule that achieves T^{opt} always exists because of the finiteness of the problem (it is always possible to enumerate all of the feasible active schedules[4] and see which are the ones that minimize $C_{\max}$). In fact, we believe that for high volume problems, there often exists more than one such schedule. As will be made evident in the next sections, finding T^{opt} is a tough combinatorial optimization problem. The optimal schedule of problems as small as 20 jobs are almost impossible to compute. Instead of finding the optimum, we use heuristics that make use of the fact that the problem is high volume, and attempt to find a solution that is almost as good as T^{opt}.

For the general job shop problem, there is no a-priori indication on which of the lower bounds is higher, the machine lower bound or the job lower bound. On the contrary, for the high volume job shop problem, the machine lower bound is higher than the job lower bound (given a large enough N). This is because as we increase N, the expected processing time on each machine grows linearly while the expected processing time of each job grows at a rate that is typically $O(\log N)$ and is bounded by $O(\sqrt{N})$ (see [15] for a description of the growth of the expected maximum of a sequence of random variables with a finite second moment). It is important to understand that a $R \ll J$ or $R \approx J$ problem of any size may be assigned processing times such that the job lower bound dominates the machine lower bound. Thus, when we say that it is the machine lower bound that dominates we are using a probabilistic argument.

[4] An *active schedule* is a feasible schedule in which no operation can be completed earlier by altering processing sequences on machines and not delaying any other operation.

1.3.2 Using the Machine Lower Bound to Estimate the Effectiveness of a Heuristic

We denote the $C_{\max}$ of a schedule that is generated by a heuristic as T^H. Remembering that T^* denotes the machine lower bound, the following holds:

$$T^* \leq T^{opt} \leq T^H \tag{1.7}$$

We denote the *gap*: $T^H - T^*$ as $\mathcal{G}$. Thus if $\mathcal{G}$ is small, we know that the sub-optimality of a scheduling heuristic is small. Note that if $\mathcal{G}$ grows in N with a rate that is less than linear ($o(N)$), then the ratio $\mathcal{G}/T^*$ tends to 0 as N increases and we may say that the heuristic is *asymptotically optimal.* This is due to the fact that T^* increases linearly in N. In this study, we analyze the rate of growth of $\mathcal{G}$, we conjecture that for some very simple heuristics it grows at a rate of $O(\log N)$ (with a high probability), while for other heuristics it is actually constant: $O(1)$.

1.4 A Survey of Job Shop Scheduling Techniques

Solving the general job shop problem is not an easy computational task, the problem is categorized as NP-complete[5]. In the early 1960's Muth and Thompson [42] produced the MT10 problem. This is an instance of the job shop problem that has 10 machines and 10 routes. Finding the optimal schedule in-terms of $C_{\max}$ for the problem served as a challenge that the scheduling community faced for many years. Only in 1989 were Carlier and Pinson [10] able to find the optimal solution for MT10. They used a branch and bound method (discussed below).

We now review some of the techniques and results regarding the solution of the general job shop problem with respect to $C_{\max}$. We start off in Section 1.4.1 where we review the disjunctive programming formulation of the problem and briefly discuss branch and bound solutions. We then continue to Section 1.4.2 which is a quick tour through some of the heuristics and approximations that have been devised. It is important to understand that all of the methods that are presented in this section are designed to solve the general job shop problem. None of the methods and results make use of properties that are found in the high volume job shop problem, specifically the $R \ll J$ case. Also note that most of the algorithms and heuristics that we describe below, have not been formulated for problems that allow re-entry but rather for problems in which the operations of each route are performed on a permutation of the machines.

[5] See [20] for a discussion of NP-completeness.

1.4.1 The Disjunctive Programming Formulation

The job shop problem may be formulated by using a weighted directed graph. Define $G = (O, C, D)$ to be a graph where O are the nodes, C are *conjunctive* arcs and D are *disjunctive* arcs. These names are part of the terminology of disjunctive programming. In this type of mathematical programming, a constraint is called conjunctive if it must be satisfied; a set of constraints is refereed to as disjunctive if at least one of the constraints has to be satisfied.

The nodes, O correspond to all of the operations that are performed on the J jobs; there is one node for each operation (r,o) and also a source node (s) and a sink node (t). The conjunctive arcs, C represent the precedence relationships between the operations on a single job/route. This means that for every two nodes, (r,o-1) and (r,o) there is a conjunctive arc in C starting at (r,o-1) and ending at (r,o). The disjunctive arcs, D connect operations that share a machine. If two nodes, say k_1 and k_2, represent operations that belong to the same machine ($\sigma(k_1) = \sigma(k_2)$) then there are two disjunctive arcs between k_1 and k_2 (facing opposite directions). The weights of each of the arcs are set to be the processing times of the corresponding nodes (the node at the start of each arc). The arcs emanating from the source node have a weight of zero. A selection of exactly one disjunctive arc from each pair such that the resulting directed graph is acyclic corresponds to finding a feasible schedule[6]. The makespan for this schedule is the length of the longest path from source to sink. See [2], [45] and [44] for details.

We now formulate the problem as a disjunctive programming problem[7]: Minimize $C_{\max}$ subject to the following constraints:

$$s_{(r,o)} \geq s_{(r,o-1)} + X_{(r,o-1)} \tag{1.8}$$

$$C_{\max} \geq s_{(r,K_r)} + X_{(r,K_r)} \quad r = 1, \ldots, R \tag{1.9}$$

$$(\sigma(r_1, o_1) = \sigma(r_2, o_2)) \Longrightarrow (s_{(r_1,o_1)} \geq s_{(r_2,o_2)} + X_{(r_2,o_2)}) \bigvee (s_{(r_2,o_2)} \geq s_{(r_1,o_1)} + X_{(r_1,o_1)}) \tag{1.10}$$

$$s_{(r,o)} \geq 0 \tag{1.11}$$

[6] It actually corresponds to finding an *active schedule*, see [45] for details.

[7] We omit j from $s_{(r,o)}(j)$ and $X_{(r,o)}(j)$. This is because in the general job shop problem there is only one job per route (route and job are synonymous).

Here constraints 1.8, 1.9 and 1.11 are conjunctive constraints. Constraint 1.10 is a disjunctive constraint. It is actually composed of two constraints for each pair of operations that share a machine. Of the two, one must be satisfied.

A solution to this problem is only obtainable by enumeration of all of the possible active schedules. Methods that a perform this enumeration are called *branch and bound* techniques. See [35] for a review of the various branch and bound algorithms that have been devised for job shop scheduling. The performance of branch and bound solutions is quite limited, even small problems are sometimes computationally impossible to solve. See [37] and [9] for benchmark times.

While most of the branch and bound techniques enumerate solutions to the disjunctive programming problem stated above, Martin and Shmoys [39] suggest a new approach: they use a time oriented branching scheme. Tests conducted using this algorithm have obtained optimal solutions or best known solutions for several benchmark problems. In addition, the time-oriented approach allows extension of the problem to incorporate additional constraints such as release dates and deadlines for jobs.

1.4.2 Heuristics and Approximations

With the absence of the ability to efficiently solve the job shop problem in realistic time, approximating heuristics are introduced.

One of the most successful heuristic procedures that were developed is the *shifting bottleneck heuristic* (Adams, Balas and Zawack [1]). The heuristic operates by using an iterative process that sequences one machine at a time[8]. Every time a new machine has been sequenced, the heuristic re-optimizes the sequence of each of the previously sequenced machines that are susceptible to improvement by again solving a one machine problem. We label the one machine problem by $P(i, I_0)$. Here i denotes the machine that is currently being scheduled and I_0 denotes the set of machines that have already been scheduled. $P(i, I_0)$ is formulated so that solving it is equivalent to minimizing the maximum lateness in a one-machine scheduling problem (for machine i) with due dates (see details in [1]). Here is an overview of the heuristic:

Step 0 - Set $I_0 = \emptyset$.

Step 1 - Identify the *current bottleneck machine*, i, among the machines $I \setminus I_0$.

Step 2 - Sequence i optimally (Solve $P(i, I_0)$).

Step 3 - Set $I_0 = I_0 \cup \{i\}$.

Step 4 - Re-optimize the sequence of each critical machine $l \in I_0$ in turn, while keeping the other sequences fixed. Do this by setting $I_0^{'} = I_0 \setminus \{l\}$ and solving $P(l, I_0^{'})$.

[8] The term "sequencing a machine" means selecting the disjunctive arcs for that particular machine.

Step 5 - If $I_0 = I$ stop, otherwise goto step 1.

The identification of the "current bottleneck machine" in step 1 is done by finding the machine that had the highest objective value in $P(i, I_0)$ (had the maximum lateness). Thus the heuristic is called shifting bottleneck. Computational comparisons have shown that for many problems, this heuristic achieves an optimum (note though that it does not produce a proof that it did so).

Williamson et al. [57] show that when the processing times are taken to be integers, deciding whether a given problem instance has a schedule that features a $C_{\max}$ of at most 3 is a polynomial problem; they also show the same thing for a length of 4 or above is an NP-complete problem. This result quickly yields the following: Finding a near-optimal schedule with a makespan of T^H such that $T^H < \rho T^{opt}$ and $\rho < 5/4$ is an NP-complete problem. This means that the best performance that may be achieved by a polynomial heuristic is bounded from below by 5/4 of the makespan of the optimal schedule[9].

The reasoning is as follows: Suppose that for some $\rho < 5/4$ we have a polynomial-time approximation algorithm that is guaranteed to produced a schedule with a makespan of at most ρT^{opt}. Now look at all of the possible problem instances. Some have $T^{opt} < 4$ while others have $T^{opt} \geq 4$. For the first type of problem instances, the approximation algorithm will yield a schedule with a length of less than $5/4 \cdot 4 = 5$. For the second type of problem instances the approximation algorithm yields a schedule with a length of at least 5. Thus such an approximation algorithm supposedly does in polynomial time what is proved in [57] to be achieved by an NP-complete algorithm. As long as $P \neq NP$, this is a contradiction.

In [51], Sevast'janov presents a survey of solutions to scheduling problems that are based on geometrical ideas[10]. For the general job shop problem, Sevast'janov presents three polynomial time approximation algorithms whose bound is independent of the number of machines (a gap of $O(1)$). We will arbitrarily refer to the algorithms as the first, second and third algorithms and refer to their gaps as $\mathcal{G}^1$, $\mathcal{G}^2$ and $\mathcal{G}^3$ respectively. We also refer to $X_{\max}$ as the processing time of the longest operation and K is the number of operations on each route (we assume constant K_r). Note that while the first and second algorithms are designed for the general job shop problem, the third is designed for the general job shop problem without re-entry. Sevast'janov shows the following:

$$\mathcal{G}^1 \leq K(I^2K^2 + K - 2)X_{\max}$$

[9] Note that this is a statement regarding all of the possible problem instances. There exist many instances that are scheduled optimally by some heuristics such the shifting bottleneck heuristic

[10] See Sevast'janov [50] for the original paper.

$$\mathcal{G}^2 \leq (K-1)(IK^2+2K-1)X_{\max}$$

$$\mathcal{G}^3 \leq (I^3+2)(I-1)X_{\max}$$

These results are important because the gap is $O(1)$ with respect to the number of jobs. This means that for $R \approx J$ problems, an asymptotically optimal solution exists. Nevertheless, the polynomial complexity of the solution and the dependence of the gap on the number of machines, maximal processing time and the number of operations may be problematic. See [6] for a discussion and numerical comparison. Note also that Sevast'janov's results do not contradict those of Williamson et al. [57]: Sevast'janov's results are asymptotically optimal only with respect to the number of jobs.

Jansen, Solis-Oba and Sviridenko [29] present a linear time approximation scheme. Their algorithm partitions the jobs into three sets: *big*, *small* and *tiny*. The sets of big and small jobs are set so that they have constant size. The algorithm then constructs *relative schedules* for the big jobs; since the number of big jobs is constant, the total number of relative schedules is also constant. In any relative schedule for the big jobs, the starting and completion times of the jobs define a set of time intervals, into which the small and tiny jobs are to be scheduled. The algorithm uses linear programming to find a compact assignment of small and tiny jobs to these time intervals. The algorithm then uses a novel rounding technique to reduce the number of jobs that receive fractional assignments to a constant. Since only small and tiny jobs receive fractional assignments, it is possible to use a very simple rounding procedure for them to get a non-preemptive schedule without increasing the length of the solution by too much. This solution is not feasible though, since in each interval there might be conflicts among the small and tiny jobs. To achieve feasibility, the algorithm finds a feasible schedule for the small and tiny jobs in each time interval by using one of Sevast'janov's algorithms.

In [29], the authors show that by selecting properly the sets of big, small and tiny jobs, it can be proven that the total length of the schedule is at most $(1+\epsilon)T^{opt}$. All of the steps of the algorithm can be performed in linear time except two of them: solving the linear program and running Sevast'janov's algorithm. Since the algorithm does not solve the job shop problem exactly, the linear program isn't to be exactly solved. They use an approximation to the linear program. The algorithm then uses an elegant idea of merging certain subsets of jobs together to form larger jobs to decrease the running time of Sevast'janov's algorithm to $O(N)$. Thus they find an algorithm whose overall complexity is linear in N. Note though that as expected by Williamson's result, the complexity is not polynomial in $1/\epsilon$.

There exist many algorithms that use *local search* methods to approximate a solutions for the problem. The algorithm by Van Laarhoven, Arts and Lenstra [53] is just one example; this algorithm uses the method of *simulated annealing*[11]. The algorithm presented by Dell'Amico and Trubian [16] and the algorithm presented by Nowicki and Smutnicki [43] use *taboo search* methods[12]. See [28] for a discussion regarding the various tabu search, genetic algorithms and simulated annealing job shop scheduling techniques that have been explored.

All of the above procedures were designed to solve the general job shop problem. These procedures do not take into account the fact that the problem may be a high volume $R \ll J$ problem (as defined in Section 1.3). In Chapter 2 we present scheduling heuristics that attempt to approximate an optimal solution to the $R \ll J$ problem.

1.5 Real World Applications of the Job Shop Problem

As stated in the beginning of this chapter, the job shop problems formulation lacks some of the details that are required to accurately model a real world factory (transfer times, set up times, etc ...). In addition, there is another important difference between job shop problems and manufacturing scenarios that arise in industries: job shop problems are finite horizon while real world factories tend to want to continuously work. Nevertheless, studying and analyzing the basic bare models should serve as a basis for more efficient scheduling in the real world.

Applications of the general job shop problem include scheduling of operations that robots need to perform to complete a certain task and scheduling of processes in a concurrent computing environment. See [45] for a description of various other applications of the general job shop problem

Applications of the high volume job shop problem, specifically the $R \ll J$ case are quite different. Solutions of this problem may be applicable to any mass manufacturing settings in which there are many operations for each job and many jobs (products) that share a common route. The semi-conductor industry may serve as an example. A typical semi-conductor manufacturing plant (wafer-fab) features processes that require hundreds of operations. In this case, the routes usually re-enter the same machine several times. See [33] for attempts to model re-entrant lines.[13]

[11] Simulated annealing is a random optimization method that combines elements of classical descent methods and random walk optimization methods.

[12] An introduction to the concept of tabu search methods may be found in [22].

[13] See Hilton [27] for a discussion regarding some more practical aspects.

Chapter 2

On-line Scheduling Heuristics

This chapter surveys, presents and summarizes scheduling heuristics that attempt to approximate the optimal solution of the $R \ll J$ problem. All of the heuristics presented are of an on-line type (defined in Section 2.1). Some of the heuristics are based on the optimal solution to the fluid approximation of the large job shop problem (presented in Section 2.2). These are the fluid imitation algorithm (presented in Section 2.3), the Dai-Weiss fluid heuristic (presented in Section 2.4.2) and several other fluid motivated algorithms (presented in Section 2.4). We also present on-line heuristics that are simple dispatching rules. These are the random dispatching rules (Section 2.5) and the buffer priority dispatching rules (Section 2.6). Note that the terms *algorithm*, *heuristic*, *rule* and *policy* are used interchangeably throughout the chapter.

2.1 On-Line Scheduling Algorithms

A scheduling algorithm is a procedure that generates a feasible schedule based on an input job shop problem instance (see [20] for a formal definition of an algorithm). If one was to use a scheduling algorithm to determine the order of operations of a job shop, one could first execute the algorithm and then use the generated schedule to actually run the shop.

An on-line scheduling algorithm is an algorithm that continuously monitors the state of the job shop and generates scheduling commands as the job shop progresses. This means that if one was to apply an on-line scheduling algorithm to a real world (physical) job shop problem instance, one would have to continuously "consult" the algorithm while the job shop is running.

Thus, an algorithm is called on-line if it is makes scheduling decisions in accordance with the state of the shop as the job shop progresses over time. The following definitions formalize the concept of an on-line scheduling algorithm:

Busy Machine : A machine that is currently working (performing some operation) and may not be stopped.

Scheduleable Machine : A machine that is eligible and waiting for the algorithm to make a scheduling decision regarding which job it is to process next. It is required that the machine be idle (not a *busy machine*) for at least a period of 0 time units. It is also required that at least one of the machine's buffers is non-empty.

Shop State : A description of the job shop at any time instance. The description may incorporate the location of each of the jobs, the duration that has passed since each of the jobs has started processing and any other information that is relevant for the proper operation of the on-line algorithm.

Scheduling Epoch : A time instance at which the *shop state* has been altered and there exists at-least one *scheduleable machine* in the shop.

Schedule Command : A command given to a scheduleable machine regarding which job to process from the available jobs in its buffers. The command may also be of a *voluntary rest command* type, meaning not to do anything.

An on-line scheduling algorithm operates by monitoring (and possibly altering) the *shop state* at each *scheduling epoch*. At each such epoch, a *schedule command* is passed to each of the *scheduleable machines*. Once all of the *scheduleable machines* have received *schedule commands*, the on-line algorithm hibernates until the next *scheduling epoch*.

We denote any algorithm that is not of an on-line type as an *off-line algorithm*. Note that by defining a rich enough *shop state*, any scheduling algorithm may actually be formalized as an on-line algorithm. For example, consider any off-line algorithm presented in Section 1.4. Such an algorithm may be implemented as an on-line algorithm by first generating a schedule when it is initially called (at the first *scheduling epoch*) and storing it in the *shop state*. Now, when ever a *scheduleable machine* is waiting for a *schedule command*, the algorithm dispatches the command based on the schedule stored in the shop state.

The opposite is also true: every on-line algorithm may be implemented as an off-line algorithm. This may be achieved by running a simulation of the the job shop problem (as is done in this study). The output of the simulation is a feasible schedule generated by the on-line scheduling algorithm.

This on-line, off-line duality is true only when all of the details of job shop problem (processing times and such) are completely known prior to the execution of the job shop.

In practice, this may not be the case; some data may not fully be known or in some cases only processing time averages may be known. In such cases, it is clear that using an on-line algorithm is much more practical than using an off-line algorithm.

Note that in the scheduling literature, on-line algorithms are sometimes refereed to as priority dispatching rules (see [28]). See [55] for a brief discussion regarding the benefits of on-line scheduling algorithms in comparison to combinatorial optimization methods and optimization in Markov decision processes.

2.2 Fluid Models

A fluid model is a mathematical model that describes the flow of fluid through a system of buffers powered by pumps. In the context of job shop scheduling, each machine may be viewed as a pump and each queue of jobs may be viewed as a fluid buffer. In a fluid model, the jobs are not modeled as discrete entities but rather as a continuous stream. Fluid models can be used to approximate both queueing systems (see [12]) and job shop problems (see [4], [54] and [55]).

We denote the assumption that the jobs are a stream of continuous fluid as the *fluid relaxation.* The fluid relaxation is composed of the following assumptions: (1) The randomness of the job durations is evened out and disappears. (2) A machine may work on more than one job (stream of fluid) at a given instant, dividing up its effort. (3) Machines may operate even though their buffers are empty, this is achieved by a continuous flow of fluid through the machine.

In (1), we assume that that the time required to *pump* a unit of fluid through buffer/pump (r,o) is $m_{(r,o)}$. In (2) we allow machine i to divide its pumping efforts between the buffers in C_i. We denote the rate of flow out of buffer (r,o) (and through the machine $\sigma(r,o)$) at time t by $u_{(r,o)}(t)$. We denote the amount of fluid in buffer (r,o) at time t by $q_{(r,o)}(t)$. We denote the amount of *upstream fluid* of buffer (r,o) at time t by $q^+_{(r,o)}(t)$:

$$q^+_{(r,o)}(t) = \sum_{i=1}^{o} q_{(r,i)}(t)$$

We denote the initial conditions (state of the system in time 0) of the fluid model as follows:

$$\begin{array}{ll} q_{(r,1)}(0) = N_r & r = 1, \ldots, R \\ q_{(r,o)}(0) = 0 & r = 1, \ldots, R \; o = 2, \ldots, K_r \end{array} \tag{2.1}$$

In addition we impose the following constraints[1]:

[1] See [54], [6] or [7] for an explanation.

$$\begin{array}{l} q_{(r,1)}(t) = q_{(r,1)}(0) - \int_0^t u_{(r,1)}(s)ds \\ r = 1, \ldots, R \end{array} \tag{2.2}$$

$$\begin{array}{l} q_{(r,o)}(t) = q_{(r,o)}(0) - \int_0^t u_{(r,o)}(s)ds + \int_0^t u_{(r,o-1)}(s)ds \\ r = 1, \ldots, R \; o = 2, \ldots, K_r \end{array} \tag{2.3}$$

$$\begin{array}{l} \sum_{(r,o)\in C_i} m_{(r,o)} u_{(r,o)}(t) \leq 1 \\ i = 1, \ldots, I \end{array} \tag{2.4}$$

$$\begin{array}{ll} u_{(r,o)}(t) \geq 0 & q_{(r,o)}(t) \geq 0 \\ r = 1, \ldots, R & o = 1, \ldots, K_r \end{array} \tag{2.5}$$

There are several cost functions that may be minimized. Given a cost function we define the *fluid problem* as follows: find measurable $u_{(r,o)}(t)$ that maintain the constraints 2.1 through 2.5 for all t while minimizing a given cost function. The problem may also be formulated with a given *time horizon*: $T > 0$. In such cases, the constraints are to be maintained for the interval $[0, T]$. We shall refer to the functions, $u_{(r,o)}(t)$, $q_{(r,o)}(t)$ and $q^+_{(r,o)}(t)$ that optimize the fluid problem as the *fluid solution.*

Note that it is sometimes more convenient to enumerate all of the buffers (r,o) to the set of numbers $k = 1, \ldots, K$ where $K = \sum_{r=1}^R K_r$. Throughout the continuation of this report, we may refer to a buffer as either buffer (r,o) or buffer k.

2.2.1 Fluid Solution of C_{max}

In [54], Weiss showed that the fluid model for the $C_{\max}$ objective function is easily solved. Using the notation of the previous section, the $C_{\max}$ objective for the fluid model may be written as follows:

$$\min \int_0^\infty 1_{q\neq 0}(t)\ dt \tag{2.6}$$

Here $1_{q\neq 0}(t)$ is a function that equals 1 for times t during which there exists a non-empty buffer; it equals 0 for all other t.

Each of the machines $(i = 1, \ldots, I)$ must work for a duration of at least

$$T_i = \sum_{(r,o)\in C_i} m_{(r,o)} q_{(r,1)}(0).$$

Thus we obtain the machine lower bound $T^* = \max_{i=1,\ldots,I} T_i$.

It can be seen ([54]), that by using the constant flow rates

$$u_{(r,o)}(t) = \frac{q_{(r,1)}(0)}{T^*} \tag{2.7}$$

for all $t \in [0, T^*]$, the system is drained by time T^*. Since this is a feasible solution that attains the lower bound, it is an optimal solution. The amount of upstream fluid in the fluid solution obtained by the flow rates 2.7 is

$$q^+_{(r,o)}(t) = q^+_{(r,1)}(0)(1 - \frac{t}{T^*}). \tag{2.8}$$

2.2.2 Weighted Flow Time Minimization by Means of a Fluid Model

As stated in the previous chapter, the WIT objective may sometimes be more suitable than the $C_{\max}$ objective. For the fluid model and for a finite time horizon T, the WIT objective may be written as follows:

$$\min \int_0^T w' q(t)\ dt \tag{2.9}$$

Here w is a K dimensional vector of weights and $q(t)$ is the vector of buffers $q_k(t)$.

Finding the optimal solution for the fluid problem using this objective is not as trivial as for the $C_{\max}$ objective. The problem may be formulated as a *separated continuous linear program* (SCLP) problem. SCLP problems have been discussed by Pullan ([46] and [47]). A finite algorithm for solving SCLP is introduced by Weiss [56]. See [56] for a formulation of the job shop problem as an SCLP problem and a solution to the problem[2].

2.3 The Fluid Imitation Algorithm

With the presence of optimal solutions for the fluid approximation of the job shop problem (either $C_{\max}$ or WIT), it is natural to define a *fluid imitation algorithm* (FIA). Such an algorithm attempts to imitate the optimal fluid solution.[3]

The FIA has the following components:

$Q^+(t)$: A K dimensional vector where the k'th element represents the number of jobs (discrete) that still have to pass through buffer k (the number of upstream jobs).

[2] Bertsimas, Gamarnik and Sethuraman [5] propose an alternative approximation algorithm.

[3] This algorithm was previously called Greedy Fluid Algorithm (GFA) (see [6] and [7]).

$q^+(t)$: A K dimensional vector where the k'th element represents the amount of fluid that still needs to pass through buffer k (the amount of upstream fluid).

$L_k(Q^+(t), q^+(t))$: The "lag" of buffer k. This function specifies by how much the real job shop (specified by Q) is lagging behind the optimal fluid solution (specified by q) in buffer k. Note that the value of $L_k(Q^+(t), q^+(t))$ is an ordinal amount: for every time instance t, it is comparable for all k having the same $\sigma(k)$.

Tie Breaking Rule : In cases where there is more than one buffer that reaches the minimal lag, a well defined rule that breaks the tie is to be defined.

The FIA on-line scheduling algorithm is specified as follows: At each *scheduling epoch* issue a *schedule command* to each *scheduleable machine* i, in the following manner: Calculate the lag L_k for all non-empty $k \in C_i$, then select the buffer k that maximizes the lag (argmax L_k). Schedule the next job on argmax L_k. Use the tie breaking rule if needed.

Note that the *shop state* required for the FIA is Q^+ and q^+.

In all of our examples we will use a lag function of the following type:

$$L_k(Q^+(t), q^+(t)) = \frac{Q_k^+(t) - q_k^+(t)}{q_k^+(t)} \tag{2.10}$$

Below are some notes regarding the FIA:

- See [6] for a comparison of the lag function 2.10 with several other alternatives.
- The *shop state* required for the FIA is Q^+ and q^+.
- Observe that there should be some rule that decides what to do when the argmax L_k is not composed of a single buffer (a tie breaking rule).
- We call an FIA algorithm that uses the $C_{\max}$ fluid solution presented in Section 2.2.1 the $C_{\max}$ FIA. Note the fluid solution is only specified for the range $[0, T^*]$. Nevertheless, an implementation of $C_{\max}$ FIA may take longer than T^*. We thus define $q^+(t)$ to be 0 for $T^* < t$. For such *scheduling epochs*, the tie breaking rule is often used because the lag is often infinity.
- In Section 5.3 we continue discussion of the $C_{\max}$ FIA when we analyze the simulation results.

2.4 Fluid Motivated Pipelining Schemes

We shall now describe a series of results regarding *pipelining* algorithms for the $R \ll J$ problem (or slight variants of it). Pipelining (to be fully explained shortly) is a cyclic scheduling scheme that allows a high utilization of all of the machines during each cycle. In Section 2.4.1 we describe the algorithms by Boudoukh, Penn and Weiss [7]. In Section 2.4.2 we describe the results obtained by Dai Weiss [14]. Finally in Section 2.4.3 we describe the heuristics by Bertsimas and Gamarnik [4]. What makes these algorithms important is that there is no clear indication in the theory that a problem with many jobs and a small number of routes is easier to solve optimally than a problem with the same number of jobs in which all of the jobs follow different routes. Thus if we can find an asymptotically optimal scheduling rule, we will probably be able to do much better than attempting to schedule the problem using combinatorial optimization methods. The algorithms that we describe show that it is relatively easy to obtain good approximate solutions to such problems[4].

Pipelining is an abundant concept within the theory and practice of microprocessor architecture. To pipeline means to split up a job (a calculation in the context of microprocessors) into several operations; each performed at a different machine (or unit) of the factory (or microprocessor). Several jobs of the same type may now be repeatedly sequenced. Once the pipe is full (all of the operations have ready jobs), the factory (or microprocessor) may begin to cycle with each cycle length being the duration of the bottleneck operation. Now for a large amount of jobs the scheduling scheme is efficient because the bottleneck machine is working almost all of the time (it is not working during the initial and final phases of the scheduling scheme).

In itself, pipelining is not a difficult concept, nevertheless there are difficulties and challenges that are to be handled. The difficulties that arise in the computer architecture setting are sequencing, control and overhead problems (see [34] and [26]). In an operations research setting, the difficulties that arise are problems of inaccurate (or random) processing times for each operation.

It does not take a lot of imagination to see the connection between pipelining and the fluid solution for $C_{\max}$ presented in Section 2.2.1. In the fluid solution, each of the machines/pumps is operating at the rate that keeps the bottleneck machine/pump fully busy, this is what pipelining is all about.

In Figures 2.1 and 2.2 gantt charts of a simulation run of a job shop with 3 machines

[4] The algorithms that we describe are very similar: they are all pipelining schemes that are motivated by the optimal fluid solution for $C_{\max}$, presented in Section 2.2.1. It is the the theorems and corresponding proofs that distinguish the three papers that we cover ([7], [14] and [4]).

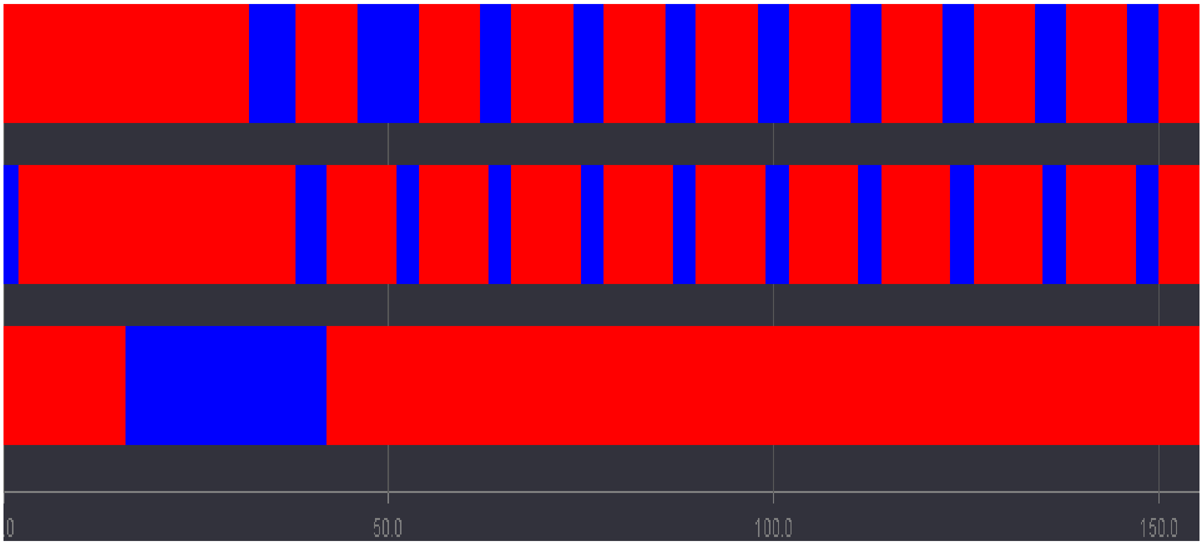


Figure 2.1: Pipelining: deterministic problem

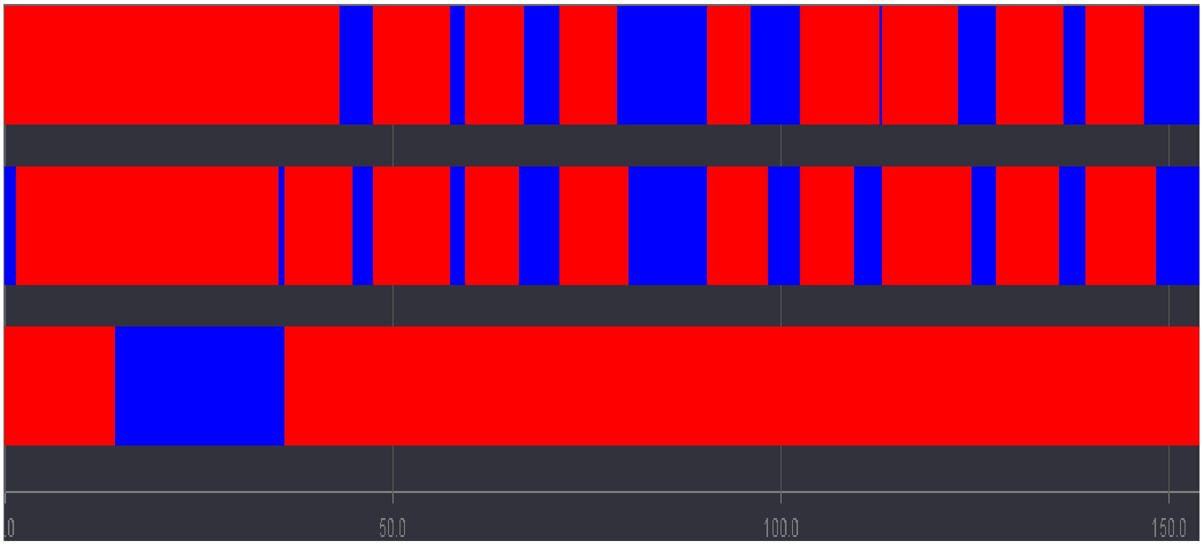


Figure 2.2: Pipelining: stochastic problem

are displayed[5]. The gantts show the execution of a job shop when scheduled by a pipelining scheme. Each of the 3 red and blue rows in the chart show the evolution of machines 1, 2 and 3 (from top to bottom) over time. A red time interval means that the machine is working, a blue time interval means that the machine is idle. For this specific problem instance, machine 3 is the bottleneck machine. Notice that except for a duration of about 20 time units near the start of the schedule, the bottleneck machine is constantly working.

Until a time of a bit over 50, the pipeline is still being "filled", at time 50, the job shop began to cycle such that the bottleneck machine is constantly working. Notice that in the deterministic problem (Figure 2.1) during each cycle, machines 1 and 2 work for the exact same time intervals. For the stochastic problem (Figure 2.2), the processing times are random and thus the duration of work within each cycle is random. Thus in the stochastic case there is a possibility that the bottleneck machine will "starve". In Section 2.4.2, we summarize an important probabilistic result that states that the probability of the bottleneck machine starving is small for high volume job shop problems.

2.4.1 Pipelining the $R \ll J$ Proportional Problem with C.V.=0

In [7], Boudoukh, Penn and Weiss describe a scheduling algorithm for the $R \ll J$ proportional problem with deterministic (C.V.=0) processing times[6]. The formulation of the algorithm assumes that there are integer constants, $p_1, \ldots, p_r$ such that $N_r = p_r N$. The formulation also adds operations with 0 processing times at the end of some of the routes such that all routes have the same number of operations $K = \max_r K_r$. Since all of the jobs that share an operation (r,o) have an identical processing time for that operation we can denote $X_{(r,o)}(j) = X_{(r,o)}$. A *cycle time* C, is defined by

$$C = \max_{1 \leq i \leq I} \sum_{(r,o) \in C_i} p_r X_{(r,o)}$$

The algorithm defines a *complete cycle* to be a cycle in which each operation (r,o) is performed on exactly p_r jobs. Thus the duration of a complete cycle is C. A *partial cycle* is defined such that if at the initial time of the cycle, t, $Q_{(r,o)}(t) < p_r$ then operation (r,o) is not performed throughout the cycle.

The algorithm now operates by performing $K - 1$ partial cycles that build safety stocks, then performing $N - (K - 1)$ complete cycles that utilize and maintain the safety stocks and finally performing $K - 1$ partial cycles that empty the safety stocks.

[5] These were generated by the JSS (see the next chapter).

[6] Several other algorithms are also described in [7].

Since the number of partial cycles at the start and the end of the schedule is independent of N, and since the bottleneck machine is fully utilized throughout the $N - K + 1$ partial cycles the heuristic's gap is $O(1)$[7].

2.4.2 Probabilistic results: The Dai-Weiss Fluid Heuristic

Dai and Weiss [14] treat the $R \ll J$ problem with random processing times and a single bottleneck machine. They assume that the processing times of jobs at each operation (r,o) are independently identically distributed with some general distribution $F_{(r,o)}$. The analysis in [14] also assumes that $F_{(r,o)}$ possesses exponential moments (the MGF exists in an interval around 0).

The authors present an algorithm for a *kitted* job shop problem. This is a job shop problem in which all of the routes have been concatenated into a single long re-entrant route. The kitted problem has the same machine lower bound as the original problem. Since the kitted problem is the original problem plus an addition of constraints, any solution to the kitted problem is also feasible for the original problem.

Here is the algorithm:

> *Phase 1* : Creation of safety stocks of S_k jobs in buffer k, for all of the buffers. S_k is of order $O(\log N)$.
>
> *Phase 2* : Performing N' complete processing cycles such that $N' = Q_1(t_f)$ where t_f is the completion time of phase 1. In each cycle, each machine performs one operation on each of its buffers. The bottleneck machine performs the processing of the operations by a cyclic order such that each operation starts as early as possible (the bottleneck machine is only idle when the next buffer according to the cyclic order is empty). All other machines start operation on cycle l, not before the bottleneck machine has started the l'th cyclic cycle. In this manner, the bottleneck machine sets the pace.
>
> *Phase 3* : After the completion of the N' complete cycles, it is not possible to perform any additional complete cycles. At this phase, the system is to "empty" using any feasible non-idling policy.

If the time that is required to complete phases 1 and 3 is $O(\log N)$ and if during phase 2 the bottleneck machine is constantly working, then the gap is $O(\log N)$. The safety stocks of size $O(\log N)$ were designed such that the bottleneck machine hardly

[7] In theorem 3.3 in [7], the authors show that $T^H - NC \leq (K-1)C$. Note $T^* = NC$.

starves. While there is a possibility that this isn't the case (since the processing times are random), Dai and Weiss prove the following: there exist constants $c_1 > 0$ and $c_2 > 0$ such that when using constant safety stocks $S_k = \lceil c_2 \log N \rceil$:

$$\mathbf{P}\{\mathcal{G}(N) \leq c_1 \log N\} > 1 - \frac{1}{N}$$

2.4.3 Bertsimas and Gamarnik's Heuristic

Bertsimas and Gamarnik [4] propose a heuristic called the *synchronization algorithm.* U_{max} is defined as as the maximum of the workloads on all machines when the multiplicity is 1. They also use a constant Ω real value (to be defined soon). The algorithm works in time intervals of length $\Omega + U_{max}$. In each such interval, each machine is to schedule a fixed amount of jobs from route r. This amount is denoted by a_r:

$$a_r = \lceil \frac{N_r \Omega}{T^*} \rceil \tag{2.11}$$

If at the start of any interval, the number of jobs of route r at any of the machines is less than a_r then the machine is to process only the jobs that are available and idle until the start of the next interval.

It is shown that when $\Omega = \sqrt{T^* U_{max} / \max_r K_r}$ the heuristic is asymptotically optimal with a gap of $O(\sqrt{N})$. Note that while Dai and Weiss have a $O(\log N)$ gap, the Bertsimas and Gamarnik results relates to the worst case (minimax) while Dai and Weiss's approach is probabilistic. Also note that this result allows the $\sum N_r$ to increase with any arbitrary mix of $N_1, \ldots, N_R$.

2.5 Random Dispatching Rules

In this section we discuss non-deterministic on-line scheduling rules. These are the random buffer scheduling rule (RBSR) in Section 2.5.1, the proportional random buffer scheduling rule (PRBSR) in Section 2.5.2 and the random job scheduling rule (RJSR) in Section 2.5.3.

2.5.1 The Random Buffer Scheduling Rule

The random buffer scheduling rule (RBSR) is very simple: at each *scheduling epoch* and for each *scheduleable machine*, enumerate all of the non-empty buffers for the specific machine and select a buffer at random (using a equal probability for each buffer). After selecting a buffer, schedule the next job from the selected buffer.

During the design phase of the Job Shop Simulation Project (see Chapter 3) we used this easy to program rule in order to perform initial tests of the software. Surprisingly we found that it performs quite well (it appears to be asymptotically optimal) for a large class of problems, see Chapter 5 for details.

We believe that this rule should be used as a benchmark for heuristics that attempt to optimize $C_{\max}$ in the $R \ll J$ problem. It does not seem reasonable to promote heuristics that are extremely complicated and are unable to perform better than this rule.

2.5.2 The Proportional Random Buffer Scheduling Rule

The previous rule (RBSR) used equal probabilities to decide from which buffer/operation to schedule at each epoch. We now propose a rule, the proportional random buffer scheduling rule (PRBSR), that uses a probability distribution that depends on the initial amount of jobs in each route (N_r), giving a higher probability to buffer with a higher corresponding N_r. The reasoning behind the rule is this: "give buffers that belong to routes that have more operations to perform a higher chance of being selected". The implementation is as follows: At each *scheduling epoch t* and for each *scheduleable machine* calculate:

$$A = \sum_{(r,o)\in C_i \quad Q_{(r,o)}(t)>0} N_r \tag{2.12}$$

Now select a buffer at random using probability N_r/A for buffer (r,o). After selecting a buffer, schedule the next job from the selected buffer.

Note that for problem instances in which N_r are equal, this rule is identical to the RBSR rule.

2.5.3 The Random Job Scheduling Rule

The random job scheduling rule (RJSR) is as simple as it sounds. This scheduling rule is also known as service in random order (SIRO), see [44]. Under this rule, each *scheduleable machine i*, chooses a job at random (with equal probability) from all of the queued jobs. Note that for the $R \ll J$ case, this rule may also be implemented as follows: select an operation/buffer at random by using the probability

$$\frac{Q_{(r,o)}(t)}{\sum_{(r,o)\in C_i} Q_{(r,o)}(t)}$$

for buffer (r,o). Then schedule the next job from the selected buffer.

This scheduling rule is somewhat similar to a *Serve the Longest Queue Scheduling Rule* since it gives a higher probability to queues/buffers with more jobs.

Remember that in the $R \approx J$ case, the term buffer is irrelevant because each of the jobs performed on machine i, is unique. In this case the RJSR and RBSR are the same.

2.6 Buffer Priority Dispatching Rules

Buffer priority dispatching rules are a class of scheduling rules that are as simple as the random buffer scheduling rules that were discussed in the previous sections. A buffer priority dispatching rule uses a priority function

$$\pi : \{(r,o) : (r,o) \in C_i \; i = 1, \ldots, I\} \longrightarrow \mathcal{N}$$

to assign a priority to each buffer. The priorities are unique:

$$(r_1, o_1) \neq (r_2, o_2) \Rightarrow \pi((r_1, o_1)) \neq \pi((r_2, o_2))$$

The priorities are assigned to the buffers have an ordinal meaning: we say that a buffer has a high priority when π is a small integer[8]. The on-line scheduling rule compares all of the non-empty buffers on each *scheduleable machine* and schedules the buffer with the highest priority.

Two special cases of buffer priority scheduling rules are last buffer first serve (LBFS) and first buffer first serve (FBFS).

Under LBFS the priorities maintain the following:

$$\begin{array}{ll} \pi((r_1, o_1)) > \pi((r_2, o_2)) & \text{if } o_1 < o_2 \\ \pi((r_1, o_1)) > \pi((r_2, o_2)) & \text{if } o_1 = o_2 \text{ and } r_1 > r_2 \end{array} \tag{2.13}$$

Thus operations that are closer to the end of the route are given higher priority and ties are broken by using the route number. The FBFS rule does the opposite, operations that are near the start of the routes are given higher priorities:

$$\begin{array}{ll} \pi((r_1, o_1)) < \pi((r_2, o_2)) & \text{if } o_1 < o_2 \\ \pi((r_1, o_1)) > \pi((r_2, o_2)) & \text{if } o_1 = o_2 \text{ and } r_1 > r_2 \end{array} \tag{2.14}$$

Note that for the re-entrant line (a high volume job shop problem with a single route and more operations than machines), the LBFS and FBFS problems are defined using the first inequality only. These buffer priority disciplines have been studied within the context of multi class queueing networks, see [33], [12] and [13].

[8] This notation is adapted from [13].

Chapter 3

The Job Shop Simulation Project

With the lack of theoretical results (theorems) regarding answers to some of the questions that we introduced in the overview (page 1) we set out to perform a simulation study[1]. After investigating the possibilities of using professional simulation software and failing to find a software package that met our needs, we designed and implemented the Job Shop Simulation Project (JSSP), a software package designed specifically for simulation of $R \ll J$ job shops driven by on-line heuristics[2].

In this chapter we describe some of the aspects of the JSSP from a software design perspective. In Section 3.1 we describe the use cases and configurations of the JSSP. In Section 3.2 we present a short overview of the design of the JSSP. In Section 3.3 we discus some of the issues regarding the implementation of the simulation kernel. In Section 3.4 we describe how the JSSP is used to create a database of simulation results. Finally in Section 3.5 we present the Job Shop Simulator (JSS), this is the front end GUI (Graphical User Interface) that may be used for interactive experimentation of on-line algorithms for job shop scheduling.

3.1 The Use Cases and Configurations of the JSSP

The JSSP is designed for the following use cases[3]:

1. Perform large scale simulation studies regarding a wide range of on-line scheduling heuristics using a variety of job shop problem instances and varying multiplicities.

2. Act as a teaching and/or self exploring aid regarding scheduling methods of job shop problems.

[1] See [8] for an introduction to the concept of computer simulation.

[2] The JSSP may also simulate $R \approx J$ job shops, but it does this inefficiently. It can be done by treating the $R \approx J$ shop as a $R \ll J$ shop with only one job on each route and many routes.

[3] For a description of the term *use case* and other software design terms, see [18].

3. Allow for neat visual presentation of new scheduling heuristics in-front of both professional and dilettante audiences.

4. Allow for visual demonstration of scheduling heuristics through the Internet.

The JSSP has two components: the *simulation kernel* and a *front end* (there are several types of front ends). The simulation kernel is the engine that performs the discrete event simulation of job shop problems. The front end is the application that may be used by users to explore the dynamics of job shop problems graphically or to control the simulation experiments. The front end is available in several configurations, one of which is the Job Shop Simulator (JSS) (see Section 3.5 for details) another is a simple batch application (not covered in this chapter) and a third is a Mathematica notebook that is connected to JAVA through J/Link (see [58] and Section 3.4).

The JSSP is packaged in three configurations:

1. JAVA Classes that are not specifically bound to any certain application. These are the simulation kernel classes. These classes may be used by any "driver" application. They are used for "heavy duty" simulation of job shops. These classes constitute the simulation kernel and may be bound to any front end.

2. A stand alone application, the JSS. This application may be easily installed and run on almost any machine, Linux, MS-Windows or Solaris. It is equipped with a robust GUI. It is not designed to be used for multiple heavy duty simulations but rather for exploratory and demonstration purposes.

3. An applet (this is also the JSS in applet form). An applet is a JAVA program running inside an Internet browser. The applet's GUI is almost identical to the application's GUI. It runs through an Internet site, and may easily be configured by a web-site builder to sit on any Internet page.

All three configuration may be down-loaded or run from this Internet site:

`http://rstat.haifa.ac.il/~yonin/thesis/jobshopsim/shopsim.html`

3.2 An Overview of the Design of the JSSP

In this section we present an overview of the implementation of the software. Note that an understanding of this section may require some knowledge of both object oriented design (OOD) and the JAVA language. For an introductory book on both subjects, see [17]. For a more advanced book regarding OOD and the UML (Unified Modeling Language) see [18]. For a classic book on design patterns see [19].

The JSSP is designed and coded in a completely OO manner using the JAVA language and JAVA API (see [23], [17] and [21]). Several reasons were the driving force for writing the code in JAVA (as opposed to C++):

1. JAVA allows for extremely fast application development.

2. JAVA allows for multi-platform compatibility.

3. JAVA allows for applets.

The main draw back in JAVA is that it is supposedly slow, (it is interpreted rather than compiled). Nevertheless it should be noted that the computational intensive sections of the software are actually compiled during runtime (using JAVA Hot-Spot).

The JSSP spans around 100 classes and interfaces. These classes are packaged in several packages:

- `package haifa.shopsim` - Contains the fundamental classes and interfaces that define the framework of the simulation.

- `package haifa.shopsim.fastkernel` - Contains the classes that implement the simulation kernel[4].

- `package haifa.shopsim.lab` - Contains R.V. generation, statistics collection, and batch execution classes.

- `package haifa.shopsim.algorithms` - Contains implementations of specific scheduling algorithms.

- `package haifa.shopsim.UI` - Contains, the classes that implement the GUI of the JSS.

- `package haifa.shopsim.UI.shopanim` - Contains the classes that are relevant for the animation of the job shop.

- `package eduni.simdiag` - Contains the gantt chart classes (imported from Fred Howell's *simjava*, see [41]).

[4] This package is the successor of the `haifa.shopsim.kernel` package that was used in earlier versions of the JSSP, see Section 3.3 for details.

We now expand on the contents of the `haifa.shopsim` package (the other packages are not covered in this section). The `haifa.shopsim` package contains the basic building blocks of the JSSP. It defines the following classes/interfaces:

- `interface ShopData` - The information regarding the job shop problem being simulated. Specifies the routes of the job shop and the mean processing times in each of the operations on each route. There are several classes that implement this interface. Some are classes that read the shop data from a text file using a predefined format, the `.jbs` format (see Appendix C for a specification). Others are used to automatically generate random job shop problem instances (such as $R \approx J$ problems).

- `interface ShopState` - The information of the job shop being modeled at a certain point in time. Specifies, the simulation time, the state of each of the machines (idle or working on a job from a certain buffer), $Q(t)$ and $Q^{+}(t)$. This class is part of a scheme that uses the observer design pattern (as specified in the Design Patterns book [19]). As specified in the observer pattern it is the *subject*; it notifies observers (statistics collectors and such) whenever changes to the state occur.

- `interface ShopAlgorithm` - Looks at the `ShopState` and decides what to schedule. This interface uses the command design pattern [19] with the `whatNow()` method as *execute()*.

- `interface ShopSimulation` - Maintains an event queue and data structures that represent the job shop (see [49] for a review of discrete event simulation techniques). Has the responsibilities of starting and stopping the simulation, setting the multiplicity of the problem and selecting a random number generator. Note that a single class may implement both this interface and the `ShopState` interface (such is the `FastShopRun` class that is explained in Section 3.3).

- `class ShopCommand` - Returned by an algorithm to the `ShopSimulation` whenever the `ShopSimulation` queries the algorithms regarding what to schedule. (This is the return type of the `whatNow()` method). The class holds the information of the *schedule command* as specified in the definition of an on-line algorithm in Section 2.1.

In order for a simulation run to occur, instances of classes that implement the above interfaces are to be created and properly linked (this is the responsibility of the front end which may be either GUI, a batch application or Mathematica). As the simulation runs, the `ShopSimulation` object advances the simulation time in discrete steps, jumping from event to event (*scheduling epoch* to *scheduling epoch*). As the time progresses, the `ShopState` object is updated. At every scheduling epoch the `ShopAlgorithm` is used to decide what to schedule on each *scheduleable machine*. The decision that the shop algorithm makes, is based on the `ShopState` (the `ShopAlgorithm` has a reference to it). The scheduling command is then returned via a `ShopCommand` to the `ShopSimulation`. The `ShopSimulation` interprets the command, and updates the state. When the `ShopState` is updated, it notifies all of its listeners (observers); these may be statistics collectors, animations or gantt charts. The process continues in this manner until the simulation is complete.[5]

The above classes/interfaces are the building block of the simulation environment and are mandatory for a simulation to occur. In addition the `haifa.shopsim` package also defines the following:

- `interface ShopChangeListener -` This interface is implemented by whatever class is interested in listening to changes of the shop. It is part of a design that uses the observer design pattern [19]. The classes that implement this interface are the statistics collection classes and the graphics visualization classes. Thus as the simulation runs, every change to the `ShopState` (taking the role of the *subject* as specified by the observer pattern) causes a notification to the listeners. Each `ShopChangeListener` is notified via the `shopChanged()` method.

- `class ShopChangeEvent -` Fired to all `ShopChangeListener` objects when the state of the `ShopState` changes. This is the parameter of the `shopChanged()` method that is specified by `interface ShopChangeListner`.

- `interface PostRunAction -` Specifies the `doAction()` method (uses the command design pattern). This method is invoked when the simulation is complete. It may do one of several actions: save data to disk, print results to screen etc...

The above is only a shallow summary of the backbone of the JSSP, for more in-depth information refer to the source and its documentation in Appendix D. It should be noted that the above specifies a framework for simulation of a manufacturing system that is not necessarily a job shop. If one was interested to transform the JSSP into simulation software of an open shop or a DAG shop, the same framework may be used.

[5] It should be noted that this is very similar to the definition of an on-line algorithm as presented in 2.1.

3.3 Implementing the Simulation Kernel

We shall now outline the manner in which the JSSP's simulation kernel operates. The discussion will focus on the two important classes that are used: `EventQ` (Section 3.3.1) and `FastShopRun` (Section 3.3.2); both are from the `haifa.shopsim.fastkernel` package. A review of some of the concepts of discrete event simulation may be found at [8], [11], [24], [30] and [49].

Before describing the implementation of the `haifa.shopsim.fastkernel` package, it should be noted that it is a "second attempt". It was only created after extensive experimentation with its predecessor, the `haifa.shopsim.kernel` package. The `haifa.shopsim.kernel` package used a set of simulation classes written by Helsgaun [25] and packaged in a package named `javaSimulation`. Helsgaun's package uses multiple threading to implement co-routines that allow to program using process-based discrete event simulation techniques in JAVA. Helsgaun based his work on the SIMULA programming language and the discrete event simulation capabilities that it offers (see [8] for a review of SIMULA). While it was very straight forward to program using `javaSimulation`, the simulation ran at an extremely slow speed (compared to the `haifa.shopsim.fastkernel` package). The slow speed is clearly attributed to the high overhead caused by thread synchronization that is handled by the JVM (JAVA Virtual Machine) when executing multiple threads. In [25], Helsgaun explains how he implements the co-routine classes that allow the user to use a process-based technique and why the process based approach is so slow.

3.3.1 The Event Queue

The `EventQ` class from the `haifa.shopsim.fastkernel` package implements a simple queue data structure. The queue is implemented by using a bi-directional linked list where each node (represented by an `Event` object) designates a simulation event. The data that is stored in an `Event` is a machine number (`int`) and a time (`double`). Each event signifies that something is due to occur at a specific machine at the specified time. If the `EventQ` is empty (has no events), then the simulation is complete. Note that the `EventQ` may be inhabited by several events with the same time as well as several events with the same machine number.

We now briefly discuss the important methods that are supplied by `EventQ`. These methods are put to use by the `FastShopRun` class as specified in the next Section:

- `double getNextTime()` - Returns the time of the next `Event`. Returns `-1.0` if the `EventQ` is empty.

- double getMachineTime(int i) - Returns the earliest time that machine i is scheduled. Returns -1.0 if the machine is not scheduled.

- int [] getNextMachines() - Returns an array of machine indexes representing the machines that are scheduled at the next earliest time. Often, this array contains only one machine index. Returns null if the EventQ is empty. In addition to returning the machine indexes, invocation of this method removes all of the events with the next time.

- void scheduleMachine(int i, double t) - Schedules a new Event. The Event is scheduled for machine i at time t.

- void removeMachine(int i) - Removes all of the events for machine i.

- void removeMachine(int i, double ut) - Removes all of the events for machine i that have a time that is smaller or equal to ut.

3.3.2 Implementing the Heart of the Simulation Kernel

We now discuss the implementation of the FastShopRun class. This class is the "heart" of the simulation kernel because it contains the logic and data structures that maintain a correct image of the job shop as the simulation runs. Below is a skeleton of the class's source code:

```
import haifa.shopsim.*;

public class haifa.shopsim.fastkernel.FastShopRun
   implements haifa.shopsim.ShopSimulation, haifa.shopsim.ShopState
{
   ...
   protected ShopData shopData;
   protected ShopAlgorithm shopAlgorithm;
   protected haifa.shopsim.lab.ProblemSizeChooser problemSizeChooser;
   protected haifa.shopsim.lab.RandomTimeMaker randomTimeMaker;
   ...
   protected EventQ eventQ;
   private Machine [] machines;
   private Buffer [] firstBuffers;
   private int [] finishedJobs;
   ...
   public void go(){...}
   private void setUp(){...}
   private void updateMachines(int [] indexes){...}
   private void performSchedulingCommand(ShopCommand sc){...}
   ...
   private class Machine{...}
   private class Buffer{...}
}
```

As may be seen, the FastShopRun class implements both the ShopSimulation interface and the ShopState interface. The go() method is the important method that is specified by the ShopSimulation interface and we discuss its implementation below (it makes use of the three private methods: setUp(), updateMachines() and performSchedulingCommand()). Each call to go() by the front end, causes another simulation run to start. While we do not discuss the details of the implementation of the ShopState interface, it should be noted that there are dozens of data members and methods involved.

As may be seen, there are references to a ShopData object and to a ShopAlgorithm object as explained in Section 3.2. There are also references to a ProblemSizeChooser and RandomTimeMaker objects. These are used to select the initial number of jobs on each route (at the start of each simulation run) and to generate random processing times (every time a job is scheduled) respectively. It should be noted that the method specified by class RandomTimeMaker to get the processing time is getTime(double mean).

There is also a reference to an EventQ class. The EventQ is the "god" that tells the simulation what should happen next and what is the current time (see the previous section). There are two types of inner classes defined: Machine and Buffer. For each machine in the job shop, the FastShopRun keeps a reference to Machine object in the array machines. For each route, it keeps a reference to the first buffer in the route in the array firstBuffers (the buffer is represented by a Buffer object). Finally for each route there is an entry in the array finishedJobs. The values in this array specify how many jobs have passed through the corresponding routes.

As the FastShopRun object is constructed, $\sum_{r=1}^{R} K_r$ Buffer objects are created and I Machine objects are created. We do not specify the details of these inner classes here but we do mention that all of the Buffer objects on each route are linked to each other in a linked-list fashion. In addition all of the buffers that belong to a certain machine are referenced by that machine. Let us also mention that each Buffer object has an int numInQ data member; it counts the number of jobs in the buffer[6].

[6] The FastShopRun class maintains the same information in many places (it is not minimal). This is to allow for faster execution. For example, it contains a java.util.HashMap object that maps each buffer to an integer that specifies the number of jobs in this buffer (this information is redundant given the discussed numInQ data member but is still useful). See the source code and its documentation for more details.

We now discuss the operation of the `go()` method. A skeleton of its implementation is given below (many details are omitted):

```
1  public void go(){
2     setUp();
3     int [] wakingMachines=null;
4     ShopCommand sc=null;
5     double time;
6     while(true){
7        time=eventQ.getNextTime();
8        if(time==-1.0)
9           break;
10       wakingMachines=eventQ.getNextMachines();
11       updateMachines(wakingMachines);
12       for(int i=0;i<wakingMachines.length;i++){
13          sc = shopAlgorithm.whatNow(wakingMachines[i]);
14          performSchedulingCommand(sc);
15       }
16    }
17    postRunAction.doAction();
18 }
```

It may be seen that the `go()` method enters a loop at line 6 that is only exited when the `eventQ` is empty (see previous section). Each iteration of this loop (lines 6-16) implies a discrete jump in the simulation time. In line 10 the next machines are received from the `eventQ` and in line 11 the corresponding machines are updated by the private method `updataMachines()`. The inner `for` loop in lines 12-15 initiates calls to the `shopAlgorithm`'s `whatNow()` method. The command is executed and the data structures of the `FastShopRun` object are updated in the private `perfromSchedulingCommand()` method. See the source code for the implementation of the private methods.

In addition to the main `while` loop (lines 6-16), we should note the call to the `setUp()` method prior to the loop, and the call to the `doAction()` method after the loop. The `setUp()` method ensures that the data structures of the simulation are set up properly at simulation time 0.0. This includes communication with the `ProblemSizeChooser` object that enables selecting and and setting the number of jobs on each route (N_r) accordingly. The call to the `doAction()` method which is made after there are no more simulation events, is in accordance with the discussion of the `PostRunAction` class in Section 3.2.

Note that the `FastShopRun` class implements the `ShopState` interface and thus acts as a *subject* of the observer pattern. This implies that throughout the `go()` method and the private methods that it uses, `ShopChangeEvents` are fired at all of the listeners. See the source code for these omitted details.

3.4 Collecting and Recording Results in the Simulation Results Database

As specified by the first use case in Section 3.1, the JSSP is designed to perform large scale simulation experiments. We now discuss the details. In Section 3.4.1 we describe how the `RichStatisticsCollector` class is used to collect statistics while the simulation runs. In Section 3.4.2 we describe the format in which the `RichShopStatistics` object that is generated by the `RichStatisticsCollector` is written to a database. The information presented here serves as a preview to Chapter 4 in which the actual simulation runs that were performed are described.

When one wants to use the JSSP to perform a simulation experiment, a proper front end is required. For the purpose of this study, we used Mathematica (see [58]) as a front end. By using J/Link, a Mathematica add-on that allows connectivity to a running JVM, we were able to instantiate objects from all of the required classes and control the simulation by means of a Mathematica notebook. We then used Mathematica utility functions to write the results in a convenient format to the simulation results database (as specified in Section 3.4.2)[7].

3.4.1 Collecting Statistics

The `RichStatisticsCollector` is a `ShopChangeListener` that collects data regarding the execution of simulation runs. This means that as the shop runs, `ShopChangedEvents` are fired by the `ShopState` and received by the `RichStatisticsCollector`.

The `RichStatisticsCollector` monitors the execution of a run from start to finish and generates an instance of a `RichShopStatistics` at the end of each run. Both the `RichStatisticsCollector` and the `RichShopStatistics` classes belong to the `haifa.shopsim.lab` package[8].

We first describe the data that is written in a `RichShopStatistics` object and then describe how the `RichStatatisticsCollector` uses the information gathered from incoming `ShopChangedEvents` to monitor the execution of the shop. This is the data that is stored a `RichShopStatistics` object[9]:

- *Machine Finish Times* - For each machine, this is the time at which it finished all

[7] We also used an additional/alternative front end, the `BatchRunner` class from the `haifa.shopsim.lab` package (not documented in this Chapter).

[8] The word `Rich` prefixes the names of these classes because they derive from more basic classes that have the same names without the `Rich` prefix and limited functionality.

[9] There is also some additional data that is stored in `RichShopStatistics` objects. It is data that is mainly used for debugging.

of the operations that it had to perform (from this time onward, the machine is idle). Note that the maximum of these times is equal to $C_{\max}$.

- *Total Flow time* - This is the WIT objective, calculated with a weight of 1.
- *Machine Work Times* - For each machine, this is the time during which it was actually working.
- *Machine Rest Times* - For each machine, this is the time during which it was idle.
- *Machine Starve Times* - For each machine, this is the starve time (as described in Section 1.2.1).
- *Machine Voluntary Rest Times* - For each machine, this is the voluntary rest time time (as described in Section 1.2.1).
- *Class, Algorithm and Multiplicity* - Information that describes how the run was configured. This includes a name of the `.jbs` file, the type of algorithm, the type of R.V. used and the N_r values.

The `RichStatisticsCollector` class updates the values for the above data as the simulation executes. With every `ShopChangedEvent` that is received, more "clues" are revealed and more is known. These classes are all subclasses of the abstract class `ShopChangedEvent`. They are fired by the `ShopState` during the execution of the simulation:

- `ShopStartedEvent`
- `ShopFinishedEvent`
- `MachineStartedEvent` (specifies machine).
- `MachineFinsihedEvent` (specifies machine).
- `JobFinishedEvent`
- `WillingRestStartedEvent` (specifies machine).

The names of the classes are clear enough such that they explain when each of them is fired[10]. The first event that is fired in every run is a `ShopStartedEvent` and the

[10] The `WillingRestStartedEvent` is fired whenever the `ShopAlgorithm` issues a *voluntary rest command.*

last event is the `ShopFinishedEvent`. For each machine, the `MachineStartedEvent` and `MachineFinishedEvent` come in consecutive matching pairs. Thus by listening to these events, the `RichStatisticsCollector` class has enough information to generate a `RichShopStatistics` object that contains the information that was described above.

3.4.2 The Simulation Results Database

We now describe the *simulation results database* (SRDB). This is a collection of text files that spans more than 30 MB and contains results of nearly 50,000 simulation runs[11] that were performed (we enumerate these runs in Chapter 4). After executing each of the simulation runs, the contents of the `RichShopStatistics` object that was created by the `RichStatisticsCollector` was written to the SRDB.

The SRDB is an amorphous collection of text files that contain *run entries*. Each run entry describes the results of a single simulation run. The run entries are not ordered in any particular fashion, they simply exist in all of the text files: entry after entry. The format of each entry is identical to the format of a Mathematica *list* (see [58]): it begins with the '`{`' character and ends with the '`}`' character and its elements are comma separated. The text files simply contain a continuous list of entries (not comma separated): `{` entry `}` `{` entry `}` . . . `{` entry `}` (there is no problem in concatenating all of the text files into a single large file). Each entry is a list that is composed of four comma separated components: `{`“`BeginObs`”`,`TN`,`PROB`,`DATA`}`.

The first component is simply the ”`BeginObs`” string. The second entry (labeled TN: test number) contains a number that may help identify the run. This number is generated by the front end that writes into the simulation database. The third entry (labeled PROB) is a list that specifies the name of the `.jbs` file that was used, the name of the random number generation class that was used, and the name of the algorithm that was used. The fourth entry (labeled DATA) is a list that specifies the following: N_r, work times, $C_{\max}$, WIT, bottleneck machines, last working machines, finish times, rest times, starve times, voluntary rest times and run-out times. Note that some of the entries within the DATA list are lists themselves.

[11] The term simulation run refers to an execution of the job shop problem from time 0 to $C_{\max}$.

3.5 The Job Shop Simulation Project's Front End GUI: The Job Shop Simulator

The Job Shop Simulator (JSS) is a graphical front end for the JSSP. In this section we briefly discuss the features of the JSS. We do not discuss its design and implementation. More information regarding the JSS's design and implementation may be obtained by looking at the sources and reading the documentation of the packages `haifa.shopsim.UI` and `haifa.shopsim.UI.shopanim` in Appendix D.[12]

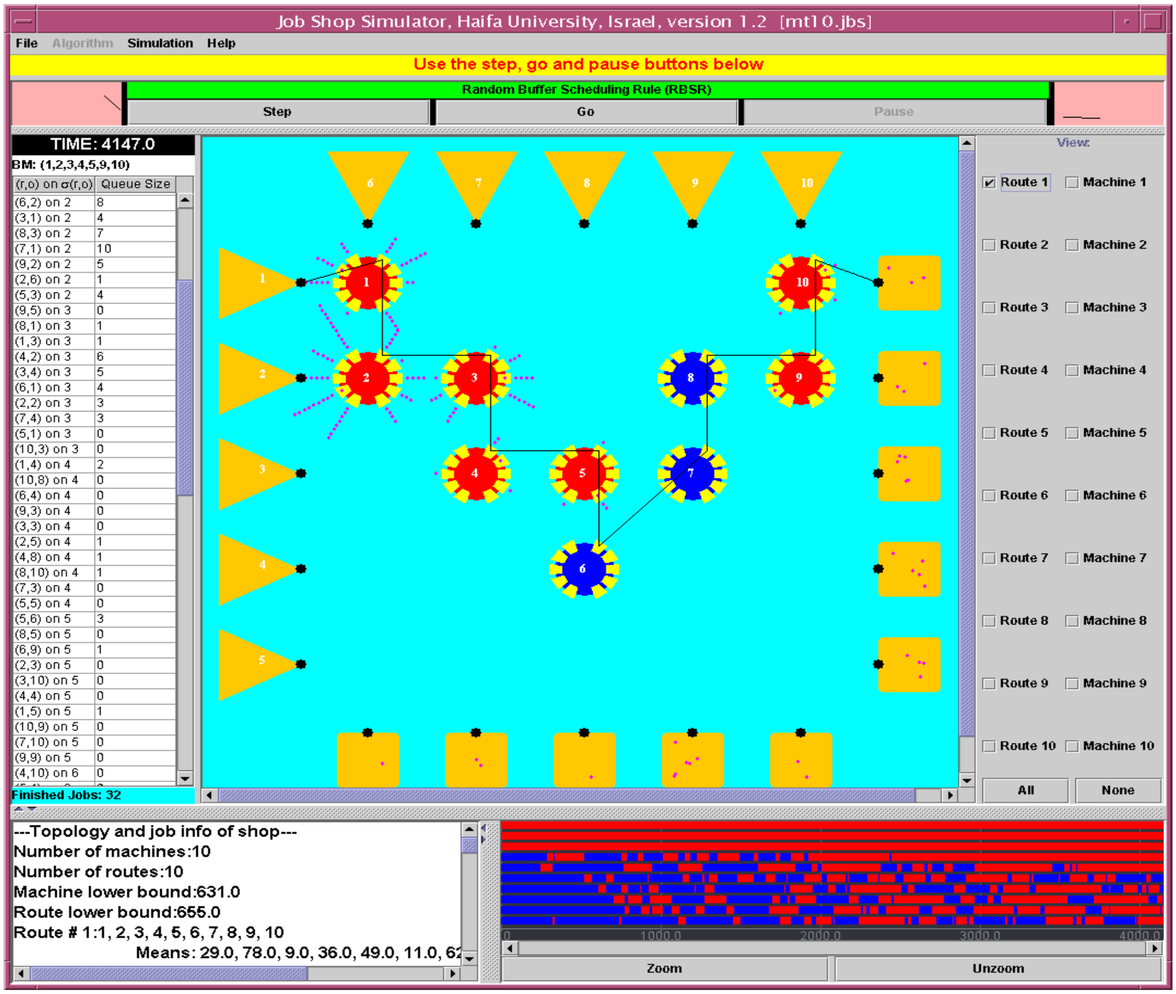
The JSS is an application that may run on any machine that supports the JAVA 1.3 Run Time Environment (JRE 1.3). It may also run as an applet. It allows the user to load a `.jbs` files and simulate the job shop problem that is specified by the file. (To create the `.jbs` file, the user should use any text editor). Several of the on-line algorithms described in Chapter 2 are supported. These are the $C_{\max}$ FIA, RBSR, PRBSR, RJSR, FBFS and LBFS. In addition to these computer controlled algorithms, a *user controlled algorithm* is also available (see Section 3.5.1).

The JSS allows the user to set the number of jobs on each route and select the processing time distribution interactively. It also allows the user to control the manner in which the simulation time advances by setting the simulation speed or using the Step Button to advance from event to event. When each simulation run is complete, the collected statistics (see Section 3.4.1) may be viewed in an auxiliary window: the Kernel Log Window.

As the JSS interacts with the simulation kernel that performs the simulation, the state of the job shop is displayed on screen: A textual table shows the sizes of the queues of each of the buffers, a graphical image represents a schematic of the job shop and a gantt chart is continuously updated. See the screen shot in Figure 3.1.

Instructions for using the JSS and interpreting the on-screen schematic animation are posted in Appendix C.

[12] The `haifa.shopsim.UI` and `haifa.shopsim.UI.shopanim` packages were implemented using Swing (A GUI toolkit supplied by JAVA). For a complete reference guide regarding graphical programming using Swing see [21].



Use the step, go and pause buttons below

Random Buffer Scheduling Rule (RBSR)

TIME: 4147.0

BM: (1,2,3,4,5,9,10)

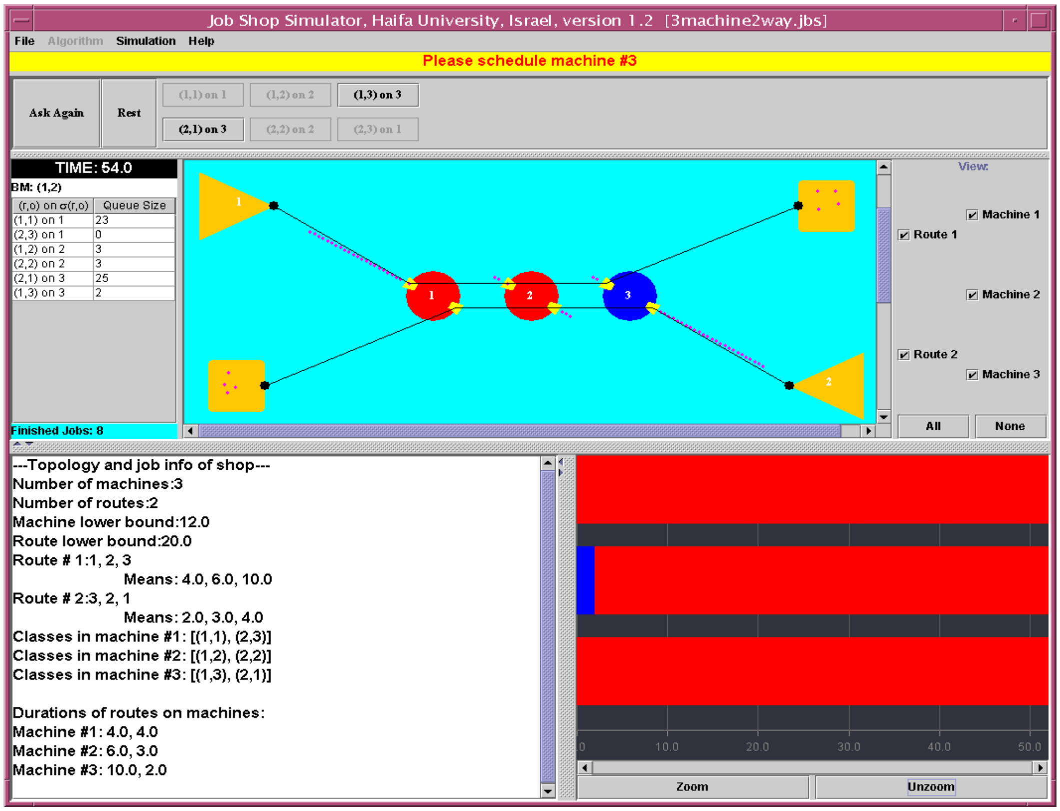
(r,o) on σ(r,o)	Queue Size
(6,2) on 2	8
(3,1) on 2	4
(8,3) on 2	7
(7,1) on 2	10
(9,2) on 2	5
(2,6) on 2	1
(5,3) on 2	4
(9,5) on 3	0
(8,1) on 3	1
(1,3) on 3	1
(4,2) on 3	6
(3,4) on 3	5
(6,1) on 3	4
(2,2) on 3	3
(7,4) on 3	3
(5,1) on 3	0
(10,3) on 3	0
(1,4) on 4	2
(10,8) on 4	0
(6,4) on 4	0
(9,3) on 4	0
(3,3) on 4	0
(2,5) on 4	1
(4,8) on 4	1
(8,10) on 4	1
(7,3) on 4	0
(5,5) on 4	0
(5,6) on 5	3
(8,5) on 5	0
(6,9) on 5	1
(2,3) on 5	0
(3,10) on 5	0
(4,4) on 5	0
(1,5) on 5	1
(10,9) on 5	0
(7,10) on 5	0
(9,9) on 5	0
(4,10) on 6	0

Finished Jobs: 32

☑ Route 1 □ Machine 1
□ Route 2 □ Machine 2
□ Route 3 □ Machine 3
□ Route 4 □ Machine 4
□ Route 5 □ Machine 5
□ Route 6 □ Machine 6
□ Route 7 □ Machine 7
□ Route 8 □ Machine 8
□ Route 9 □ Machine 9
□ Route 10 □ Machine 10

---Topology and job info of shop---
Number of machines:10
Number of routes:10
Machine lower bound:631.0
Route lower bound:655.0
Route # 1:1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Means: 29.0, 78.0, 9.0, 36.0, 49.0, 11.0, 62

Figure 3.1: The GUI of the JSS



Job Shop Simulator, Haifa University, Israel, version 1.2 [3machine2way.jbs]

Please schedule machine #3

TIME: 54.0

BM: (1,2)

(r,o) on σ(r,o)	Queue Size
(1,1) on 1	23
(2,3) on 1	0
(1,2) on 2	3
(2,2) on 2	3
(2,1) on 3	25
(1,3) on 3	2

Finished Jobs: 8

```
---Topology and job info of shop---
Number of machines:3
Number of routes:2
Machine lower bound:12.0
Route lower bound:20.0
Route # 1:1, 2, 3
                Means: 4.0, 6.0, 10.0
Route # 2:3, 2, 1
                Means: 2.0, 3.0, 4.0
Classes in machine #1: [(1,1), (2,3)]
Classes in machine #2: [(1,2), (2,2)]
Classes in machine #3: [(1,3), (2,1)]

Durations of routes on machines:
Machine #1: 4.0, 4.0
Machine #2: 6.0, 3.0
Machine #3: 10.0, 2.0
```

Figure 3.2: The user controlled algorithm

3.5.1 The User Controlled Algorithm

The *user controlled algorithm* allows the user of the JSS to determine the schedule of a job shop problem in an on-line manner. As may be seen in Figure 3.2, the algorithm area (near the top of the window) contains several buttons that allow the user to control the scheduling. These are the Ask Again Button, the Rest Button and six additional buttons (one button for every operation (r,o) in the job shop).

As the simulation kernel advances from event to event, the user controlled algorithm continuously asks the user to make on-line scheduling decisions. In the figure, the user is currently being asked to schedule machine 3 at the simulation time 54.0. Since the available operations on machine 3 are (2,1) and (1,3) ,only the corresponding buttons are active and may be clicked by the user. The user can also decide to click the Rest Button and thus not schedule any job on machine 3 at the current *scheduling epoch.*

The Ask Again Button does not have a functional purpose and was only used for debugging the simulation kernel. In fact, one of the main purposes of the user controlled algorithm was to allow easy debugging and verification of the simulation kernel.

Chapter 4

The Simulation Experiments

In this chapter we describe the simulation experiments that were performed and summarize the raw results that were obtained. In Section 4.1 we discuss the conceptual view of the job shop simulation, defining the theoretical framework on which the experiments are based. In Section 4.2 we enumerate all of the classes of simulation runs that were performed, specifying specific details such as random variable generation and meaned topology selection. In Section 4.3 we chart the details regarding the number of replicates and multiplicities that were used for the experiments. Finally in Section 4.4 we summarize the results of all of the runs into a short and informative format to be analyzed in Chapters 5 and 6.

4.1 Conceptual View of the Job Shop Simulation

We shall use $\wp$ to denote an instance of a job shop problem. $\wp$ fully specifies the machines, routes, jobs and processing times of the instance. We shall use $\mathcal{P}$ to denote the population of all job shop problem instances. In the simulation study, we categorize $\mathcal{P}$ into several *classes* of problems, each specified by different attributes. We then investigate the results of applying scheduling rules to problem instances from these classes. We shall denote each such class as $\mathcal{P}_i$. The attributes that specify each $\mathcal{P}_i$ may be either probabilistic (distribution of routes and/or processing times) or deterministic (e.g. identical versus proportional problems). Note that $\mathcal{P}_i$ is not necessarily a partition of $\mathcal{P}$ because some of the attributes of $\mathcal{P}$ are probabilistic.

Since we are interested in high volume job shop problems, we partition each $\mathcal{P}_i$ into groups of job shop problems with varying multiplicities. We label each such group as $\mathcal{P}_{i,N}$. This implies that all of the job instances $\wp$ that belong to $\mathcal{P}_{i,N}$ share common characteristics that are categorized by the class i and have a multiplicity of N.

The simulation procedure involves repeated sampling of $\wp$'s from $\mathcal{P}_{i,N}$ and scheduling them using a scheduling rule. This is done for various classes i and increasing multiplicities N. We are equipped with several scheduling rules as described in Chapter 2. Denoting a scheduling rule by Υ and a schedule by $\mathcal{S}$, we shall now denote by $\mathcal{S}_{\Upsilon}(\wp)$ a schedule generated by applying a scheduling rule on $\wp$.

Note that by taking a probabilistic approach (see 1.3.1), we may treat $\mathcal{P}_{i,N}$ as a population of random job shops $\wp$, each $\wp$ being a R.V. In this case, the simulation procedure is a statistical experiment and we treat each $\wp$ as a sample from $\mathcal{P}_{i,N}$. It follows that we also treat $\mathcal{S}_{\Upsilon}(\wp)$ as a R.V. It should be noted that the randomness of $\mathcal{S}_{\Upsilon}(\wp)$ is due to two reasons. First it is random because $\wp$ is random. Secondly, it may also be random because of random behavior present in Υ (such as in the RBSR rule).

For each $\wp$ there exits an optimal schedule $\mathcal{S}_{\Upsilon^{opt}}(\wp)$ (this schedule uses the optimal scheduling rule Υ^{opt}, that always exists but is usually very hard to find). We label the $C_{\max}$ of the optimal schedule by $T^{opt}(\wp)$.

For each sampled $\wp$ there exists several lower bounds $B_1(\wp), B_2(\wp), \ldots$. These are deterministic functions of the R.V. $\wp$ and are thus R.V.'s themselves. These are the machine lower bound, job lower bound (as described in Chapter 1) or any other lower bound. It is important to note that the lower bounds are with respect to an objective function (in our case $C_{\max}$). We will denote the tightest lower bound as $B(\wp) = \max_i B_i(\wp)$. Note that we know that $B(\wp) \leq T^{opt}(\wp)$.

Defining $T^H(\wp)$ as the makespan of each $\mathcal{S}_{\Upsilon}(\wp)$ run, we label the gap by

$$\mathcal{G}(\wp) = T^H(\wp) - B(\wp)$$

For each sampled $\mathcal{S}_{\Upsilon}(\wp)$ we also measure the rest time decomposition for the bottleneck machine (or machines) as specified in Section 1.2.1. These are the starve time, run-out time and voluntary rest time; labeled $\Im(\wp)$, $\Re(\wp)$ and $\mho(\wp)$ respectively.

We denote the process of generating a random $\wp$ from $\mathcal{P}_{i,N}$, simulating a schedule $\mathcal{S}_{\Upsilon}(\wp)$ and calculating (measuring) the array $\{\mathcal{G}(\wp), \Im(\wp), \Re(\wp), \mho(\wp)\}$ as performing a *run* from the class $\mathcal{P}_{i,N}$ with a log-multiplicity of $\log N$. For each $\mathcal{P}_{i,N}$ we perform several *replicates* (see Section 4.3.1). We refer to the term *simulation experiment* as a collection of runs performed for various increasing log-multiplicities with several replicates for each log-multiplicity on a given $\mathcal{P}_i$ with a given Υ.

Note that in practice, some more information is recorded during each run, but it is currently not used for analysis (see Section 3.4). Also note that in practice, we only use the machine lower bound as a lower bound (without using the job lower bound), this is because it was the dominating bound for all problems that we tested for a $\log N$ as low as 2.

4.2 The Classes of Experiments

We shall now specify all of the classes $\mathcal{P}_i$ (all of the simulation experiments performed). Each class is categorized by the parameters of the job shop simulation: The meaned topology (to be defined below), the number of jobs on each route, the scheduling algorithm and the R.V. used to generate the processing times.

In Chapter 1 we described a coarse division of $\mathcal{P}$: the $R \ll J$ case and the $R \approx J$ case. The $R \ll J$ case is further divided to the proportional case and the identical case. A run of the type $R \ll J$ requires a *meaned shop topology*, this is a description of the routes, and the processing means of each operation. Runs of the $R \approx J$ class do not have a meaned topology, random routes are selected instead. We now describe the attributes used to categorize the classes. See Appendix B for specific technical details regarding the generation and simulation classes of experiments.

4.2.1 The Meaned Topologies

For the case $R \ll J$ we based our study on three well known bench mark job shop problems:

MT10 : The well known 10 machine and 10 routes problem; introduced by Muth and Thompson, see [42].

ABZ9 : A 20 machines and 15 routes problem used by by Adams, Balas and Zawak in [1].

SWV13 : A 10 machine and 50 routes problem used by Storer, Wu and Vaccari in [52].

Each of these problems was originally formulated as a challenge to be solved by combinatorial optimization methods (or search methods in the case of SWV13). There is no true justification for choosing the above as shop topologies except for the fact that they are well known problems. Our choices for these problems is inspired by Lorenco [37]. Lorenco conducted a computational study of several search techniques and classical methods and used the above problems (among others). A data bank consisting of more than a 100 benchmark problems may be found at the Internet site [40].

Note that while we allow re-entry in the general job shop model and do not require all routes to be of the same length as the number of machines, the above problems all contain routes whose operations are performed on a permutation of the machines (do not feature re-entry).

In addition to the above, we created three additional meaned topologies:

MT10-bal : This is a balanced, MT10 problem. This problem has the same operations as MT10 but different processing times. The processing times were modified so that the workload of each machine is equal. This problem was created by multiplying all of the processing means of operations on machine i, by a factor that normalized the machine workload to 631 (this is the machine lower bound value of the original MT10 problem).

MT10-rline : This is the MT10 problem with kitted routes. In this problem there is only one route consisting of 100 operations. This route is a concatenation of the 10 routes. (See [14] for details regarding kitting of routes).

MT10-round-bal : This is a modified MT10 problem such that each route starts on a different machine ($\sigma(r,1) = r$). It is also balanced so that the workload of all of the machines is 631. It was created by first modifying the routes of the original problem such $\sigma(r,1) = r$ and each route traverses through all 10 machines. Then the problem was balanced as was done for the MT10-bal problem.

4.2.2 The $R \approx J$ Topologies

In addition to using the meaned topologies mentioned above, we tested problems of the type $R \approx J$. For these problems we generated random topologies using the following parameters.

I : The number of machines.

a : This parameter specifies how many steps are in each of the many routes. The number of steps is drawn from a discrete uniform distribution taking values in the range of $[I - a, I + a]$.

$EM_1, \ldots, EM_I$: These I parameters are the means of the operations on each of the machines.

4.2.3 The Processing Time Distributions

We have used several types of distributions for the processing times:

C.V. = 0 : This means that random processing time generation does not occur (no variability), the mean processing time is used instead; hence all the jobs on the same route are identical.

C.V.= 0.25 : These are normal random variables with a coefficient of variation of (standard deviation divided by mean) of 0.25 truncated at 0.

C.V.= 1.0 : These are exponential random variables (the exponential distribution's standard deviation equals its mean thus its C.V. equals 1.0).

Weibull 1/2 : This is a Weibull random variable with hazard rate of $Ct^{-1/2}$. Its tail is heavy enough to not posses a MGF , yet it posses moments of all orders.

Pareto 3 : This is a Pareto distribution with the only existing moments: first, second and third.

Pareto 2 : This is a Pareto distribution with the only existing moments: first and second.

See Appendix Section B.1 for a discussion regarding the methods used for random variable generation.

4.2.4 The Multiplicity

The multiplicities N that were used were powers of 2: $\{1, 2, 4, 8, \ldots, 2^{18} = 262,144\}$ (denoted *log multiplicities*). Problems as large as $N = 2^{18}$ were only used in two simulation experiments (such runs require several hours of processing time). Most of the classes used a maximal $\log N$ of 15 ($N = 32,768$). Classes from the $R \approx J$ case, used a maximal $\log N$ of 12 ($N = 4,096$).

4.3 Enumeration of the Experiments Performed

We now summarize all of the classes of runs that were simulated. For each class, we specify all of the attributes that distinguish the class, the scheduling heuristic used and the maximal log-multiplicity. Note that for a class with a given maximal log-multiplicity, runs were performed using all of the smaller log-multiplicities.

Tables 4.1, 4.2 and 4.3 present a summary of the simulation experiments performed[1]. The columns represent the scheduling rules that were tested and the rows represent the random variables that were used to generate processing times and the meaned topologies. All of the results were recorded to the SRDB as specified in Section 3.4.

[1] Table 4.3 refers to (I, a) as specified in Section 4.2.2

$R \ll J$, Identical case

R.V.	*Meaned Topology*	$C_{\max}$ FIA	RBSR	RJSR	FBFS	LBFS
C.V. = 0	MT10	18	18	14	15	15
	ABZ9	16	15			
	SWV13	14	12			
	MT10-bal	14	15			
	MT10-rline	14	14		14	14
	MT10-round-bal	15	15			
C.V.= 0.25	MT10	16	15	16		
	ABZ9		15	15		
	SWV13	13	12			
	MT10-bal	14	15			
	MT10-rline					
	MT10-round-bal					
C.V.= 1.0	MT10	15	16			
	ABZ9	14				
	SWV13	14	12			
	MT10-bal	14				
	MT10-rline	14				
	MT10-round-bal	15	15			
Weibull 1/2	MT10	13	14			
	ABZ9					
	SWV13					
	MT10-bal					
	MT10-rline					
	MT10-round-bal					
Pareto 3	MT10	11	13			
	ABZ9					
	SWV13					
	MT10-bal					
	MT10-rline					
	MT10-round-bal					
Pareto 2	MT10	16	15		16	16
	ABZ9	14				
	SWV13	14				
	MT10-bal	14	15			15
	MT10-rline					
	MT10-round-bal	15	15			

Table 4.1: The maximum log-multiplicities of the $R \ll J$ identical simulation experiments

$R \ll J$, Proportional Case						
R.V.	*Meaned Topology*	$C_{\max}$ FIA	RBSR	PRBSR	FBFS	LBFS
C.V. = 0	MT10	14	16	15		14
	ABZ9	13	13	15		
	SWV13	13	14			
	MT10-bal	14				
C.V.= 0.25	MT10	14				
	ABZ9	13				
	SWV13	14	14			
	MT10-bal	13				
C.V.= 1.0	MT10	14	14	13	13	13
	ABZ9	13	14			
	SWV13	13	14			
	MT10-bal					
Weibull 1/2	MT10	14				
	ABZ9	13				
	SWV13	13				
	MT10-bal					
Pareto 3	MT10	14				
	ABZ9	13				
	SWV13	13				
	MT10-bal					
Pareto 2	MT10	14	14	15	13	13
	ABZ9	13	14			
	SWV13	13	13			
	MT10-bal	13	13			

Table 4.2: The maximum log-multiplicities of the $R \ll J$ proportional simulation experiments

$R \approx J$ specified by (I,a) using RJSR		
C.V.= 0.25	(10,0)	12
C.V.= 1.0	(10,0)	12
Pareto 2	(10,0)	12

Table 4.3: The maximum log-multiplicities of the $R \approx J$ simulation experiments

4.3.1 The Number of Replicates

We refer to the number of replicates of runs in a particular simulation experiment and using a particular multiplicity as the *sample size*. We use n to denote the sample size. As we conducted the experiments we used sample sizes ranging from 15 to 50. Note that the experiments were conducted during a period of about 3 month, and each simulation run that we conducted was recorded in the SRDB. Thus, there are certain simulation experiments and certain log-multiplicities for which the sample sizes are higher than others. This is because we felt that it was bad habit to discard observations only to achieve uniform sample sizes.

4.4 The Experimental Results

After performing the simulation experiments, we plotted the results in graphs; there is one graph for each simulation experiment. All of the graphs for all of the simulation experiments may be seen at Appendix A; an example graph appears in Figure 4.1. We now describe the format and content of the graphs.

The y-axis is measured in time units of the given problem instance[2]. The x-axis is the log-multiplicity. The x-axis also displays the sample size used for each log multiplicity. The interior of the graphs present the following information:

- *observations* - The gap, $\mathcal{G}(\wp)$ of each run, $\wp$, is plotted as a single tiny dark point.
- *gap line* (heavy green line)- The heavy green line represents the average gap: $\sum_{\wp} \mathcal{G}(\wp)/n$. Where the sum is over all of the observations ($\wp$) that were sampled from a given $\mathcal{P}_{i,N}$ and n is the number of such observations.
- *gap bounds* (light green lines) - The light green lines that bound the average gap are the 5% gap distribution tails. This means that 5% of the runs are above the top light green line and 5% of the runs are below the bottom green line.
- *voluntary rest time line* (purple) -The distance between the x-axis and the purple line is the average voluntary rest time: $\sum_{\wp} \mho(\wp)/n$. This distance is 0 in all of the simulation experiments because we did not use algorithms that voluntarily rest.
- *starve time line* (blue) - The distance between the voluntary rest time line (purple) and the blue line is the average starve time: $\sum_{\wp} \Im(\wp)/n$.

[2] Since we did not "normalize" the units of all of the meaned topologies, these units are not comparable between simulation experiments that are based on different meaned topologies.

- The distance between the starve time line (blue) and the gap line (heavy green) is the average run-out time: $\sum_{\wp} \Re(\wp)/n$.

Note that for problems that have more than a single bottleneck machine (the MT10-bal problem) the idle time decomposition (purple and blue lines) is performed for an arbitrary machine.

Thus the graphs primary use is to allow examination of the behavior of $\mathcal{G}$ as the multiplicity increases. This is achieved by looking at the growth of the gap line. An additional use is the investigation of the idle time decomposition (as described in Section 1.2.1). Finally, the spread between the gap bounds may be observed, this is a measure of the variability between replicates of the gap.

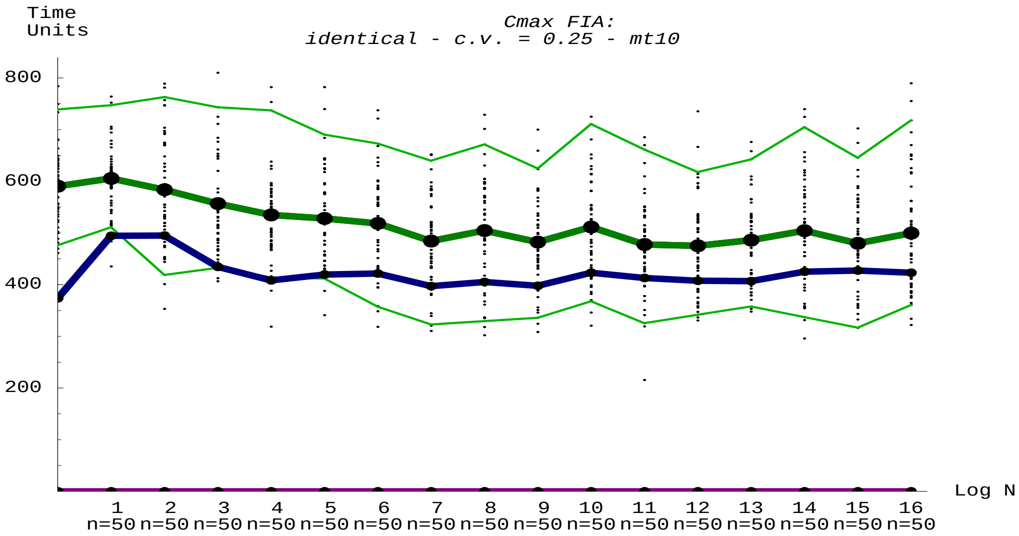


Figure 4.1: An example of a simulation experiment result

4.4.1 Summary of the Results

We now present a summary of all the results that appear in the graphs for the $R \ll J$ identical and proportional cases (we describe the results for the $R \approx J$ when we discuss them in the next chapter). We present the results of the summary in Tables 4.4 and 4.5. Each entry in the table summarizes the behavior of the gap and idle time decomposition that is observed. An entry is of the form `A/B/C`. The `A` category describes the order of the increase of the gap. We describe the various values that it may take below. The `B` and `C` categories are optional; we describe their meaning now. The `B` category may be set to the values $\Im$ or $\Re$ meaning that the either the starve time or the run-out time are the sole reason for an increase in the gap. Thus if `B` is set to $\Im$, we know that the starve time increases with N while the run-out time does not. The opposite occurs if `B` is set to $\Re$. For many results, the `B` category is omitted because both of these times increase with N or because the gap is $O(1)$. The `C` category is set to $\Im \searrow$ when the starve time is actually decreasing as the multiplicity increased; we attempt to explain this phenomena in the next chapter

We have used a simple visual inspection[3] to see how the gap varies as the multiplicity increases (category `A`). Since the graphs' x-axis is presented on a logarithmic scale in terms of the multiplicity, an exponential increase of the gap in the graph implies a linear increase with respect to N and a linear increase in the graph implies a logarithmic increase with respect to N. An increase of the gap in the graph that is faster than linear but not exponential is an increase that is slower than linear with respect to N, thus it is also an asymptotically optimal schedule.

We categorize the increase of the gap in the simulation experiments to one of the four categories: $O(1)$, $O(\log N)$, $O(\log N)+$ and $O(N)$ meaning a constant gap, a logarithmically increasing gap, a faster than logarithmic but still asymptotically optimal increasing gap and a linearly increasing gap respectively[4]. Note that any gap that is increasing less than linearly is called asymptotically optimal (as defined in Section 1.3.2). Note that in graphs where the gap is increasing faster than linearly it may be hard to make a judgment between $O(\log N)+$ and $O(N)$. For such graphs we examined the ratio $\mathcal{G}(N)/N$[5]. When the ratio approaches 0 we know that the category is asymptotically optimal and therefore $O(\log N)+$. Otherwise, the gap is $O(N)$.

[3] We did not find it appropriate to perform any more advanced statistical analysis (e.g. regression) because of the nature of the simulation experiments.

[4] These categories may be inaccurate, nevertheless they are the best that we could find by looking at the results

[5] An example of a graph of this ratio may be seen in some of the figures in the next chapter. See for example Figure 5.5.

$R \ll J$, Identical case						
		$C_{\max}$ FIA	RBSR	RJSR	FBFS	LBFS
C.V. = 0	MT10	$O(1)$	$O(\log N)/\Re$	$O(N)/\Im$	$O(N)$	$O(N)$
	ABZ9	$O(1)//\Im \searrow$	$O(\log N)/\Re/\Im \searrow$			
	SWV13	$O(1)$	$O(1)$			
	MT10-bal	$O(1)$	$O(\log N)+$			
	MT10-rline	$O(1)$	$O(\log N)$		$O(N)$	$O(1)$
	MT10-round-bal	$O(1)//\Im \searrow$	$O(\log N)$			
C.V.= 0.25	MT10	$O(1)$	$O(\log N)/\Re$	$O(N)$		
	ABZ9		$O(\log N)/\Re/\Im \searrow$	$O(N)$		
	SWV13	$O(1)$	$O(1)$			
	MT10-bal	$O(\log N)+$	$O(\log N)+$			
	MT10-rline					
	MT10-round-bal					
C.V.= 1.0	MT10	$O(1)$	$O(\log N)/\Re$			
	ABZ9	$O(1)//\Im \searrow$				
	SWV13	$O(1)$	$O(\log N)$			
	MT10-bal	$O(\log N)+$				
	MT10-rline	$O(\log N)/\Im$				
	MT10-round-bal	$O(1)$	$O(\log N)$			
Weibull 1/2	MT10	$O(1)$	$O(\log N)$			
	ABZ9					
	SWV13					
	MT10-bal					
	MT10-rline					
	MT10-round-bal					
Pareto 3	MT10	$O(1)$	$O(\log N)$			
	ABZ9					
	SWV13					
	MT10-bal					
	MT10-rline					
	MT10-round-bal					
Pareto 2	MT10	$O(1)$	$O(\log N)$		$O(N)$	$O(N)$
	ABZ9	$O(1)//\Im \searrow$				
	SWV13	$O(1)$				
	MT10-bal	$O(\log N)+$	$O(\log N)+$			$O(N)$
	MT10-rline					
	MT10-round-bal	$O(1)$	$O(\log N)/\Re$			

Table 4.4: Result summary of the $R \ll J$ identical simulation experiments

$R \ll J$, Proportional Case						
		$C_{\max}$ FIA	RBSR	PRBSR	FBFS	LBFS
C.V. = 0	MT10	$O(1)$	$O(N)/\Re$	$O(\log N)+/\Re$		$O(N)$
	ABZ9	$O(1)//\Im \searrow$		$O(\log N)//\Im \searrow$		
	SWV13	$O(1)$	$O(1)$			
	MT10-bal	$O(1)//\Im \searrow$	$O(N)$			
C.V.= 0.25	MT10	$O(1)$				
	ABZ9	$O(1)//\Im \searrow$				
	SWV13	$O(1)$	$O(1)$			
	MT10-bal	$O(1)//\Im \searrow$				
C.V.= 1.0	MT10	$O(1)$	$O(N)$	$O(\log N)$	$O(N)$	$O(N)$
	ABZ9	$O(1)//\Im \searrow$	$O(N)//\Re$			
	SWV13	$O(1)//\Im \searrow$	$O(1)//\Im \searrow$			
	MT10-bal					
Weibull 1/2	MT10	$O(\log N)$				
	ABZ9	$O(\log N)$				
	SWV13	$O(\log N)//\Im \searrow$				
	MT10-bal					
Pareto 3	MT10	$O(1)$				
	ABZ9	$O(1)$				
	SWV13	$O(1)//\Im \searrow$				
	MT10-bal					
Pareto 2	MT10	$O(1)$	$O(N)$	$O(\log N)+$	$O(N)$	$O(N)$
	ABZ9	$O(1)$	$O(N)$			
	SWV13	$O(1)//\Im \searrow$	$O(1)//\Im \searrow$			
	MT10-bal	$O(\log N)$	$O(\log N)+$			

Table 4.5: Result summary of the $R \ll J$ proportional simulation experiments

Chapter 5

Interpretation of the Simulation Results

In this chapter we analyze and interpret some of the simulation results. We begin in Section 5.1 where we discuss the results that were established prior to this study. These include the Dai-Weiss fluid heuristic results (see Section 2.4.2) and the simulation results by Boudoukh et. al. (see [7]). Continuing on to Section 5.2, we discuss a stochastic model for $0 < t < \infty$ which is the framework used to explain the performance of the $C_{\max}$ FIA for re-entrant line problems. In Section 5.3, we discuss the $C_{\max}$ FIA, describing the results that it achieved and the empirical conjectures that may be drawn. The $C_{\max}$ FIA achieved a gap of $O(1)$ in a wide variety of simulation experiments. In Section 5.4 we discuss the RBSR and PRBSR rules. The simulation results have shown that these two random scheduling rules are asymptotically optimal for a variety of problems. In Section 5.5, we briefly discuss the phenomena of decreasing starve times with respect to N. In Section 5.6 we discuss the scheduling rules that are not asymptotically optimal. These are the FBFS, LBFS and RJSR rules. Finally, in Section 5.7 we discuss the results obtained by the $R \approx J$ simulation experiments.

This chapter touches a variety of subjects in a rather informal manner. The simulation results that were presented in Tables 4.4 and 4.5 in the previous chapter contain a sea of information, some of which we interpret here. We are not yet able to prove any of these results, we rather informally discuss their meaning and attempt to give intuitive explanations. Throughout this chapter, when we say that a heuristic is asymptotically optimal it is meant that the gap is $o(N)$. We sometimes informally say that a heuristic is $O(1)$ or $O(\log N)$ etc... In this case it is meant that the simulation experiments have shown that the average gap increases at such a rate with respect to N.

5.1 Previous Results Regarding the $R \ll J$ Case

It has been shown by Boudoukh, Penn and Weiss [7], that a gap of $O(1)$ is possible to achieve for the deterministic (C.V.=0) case. In [14], Dai and Weiss generalized the problem to the stochastic case and showed that a gap of $O(\log N)$ is achievable with a high probability (increasing in N). The result by Dai and Weiss is based on an analysis of the growth of a maximum of a GI/G/1 queue. By showing that the maximum of the queue grows at a rate that is $O(\log N)$, the authors show that the gap of the heuristic is $O(\log N)$. The Dai and Weiss result is based on two assumptions: (1) The processing times come from distributions that posses exponential moments (the MGF exists at an interval around 0). (2) There is only a single bottleneck in the job shop.

While the results by Dai and Weiss supported the notion that the best gap that may be achieved for a stochastic case is $O(\log N)$, preliminary simulation results that were conducted by Boudoukh, Penn and Weiss in [7], generated a surprise. The gap that was observed for the FIA was O(1) for both C.V.=0.25 and C.V.=1.0 problems. While the maximum log-multiplicity that was used was only 10 and the number of replicates was small, these simulation results hinted that asymptotically optimal scheduling with an $O(1)$ gap are obtainable. This implies that the $C_{\max}$ FIA operates with a bounded starve time and a bounded run-out time.

5.2 A Stochastic Model and a Fluid Solution for Re-entrant Lines

We now present a stochastic model and a fluid solution of re-entrant line problems that are driven by the LBFS non-idling scheduling rule. Remember that a re-entrant line is a job shop problem in which there is only one route with K operations and $I < K$.

The Fluid Model

In the fluid model, the policy is simple: when machine i has to decide between working on buffers k_1 and k_2 then it will devote all that it can towards buffer k_2 if $k_1 < k_2$. The remainder of its effort is devoted towards buffer k_1 (or to other buffers with an index less than k_2).

We now present an iterative algorithm for calculating constant pumping rates u_k for all of the buffers $k = 1, \ldots, K$ such that the re-entrant line empties at time T^* and the LBFS scheduling rule is used. The algorithm performs the calculation in L iterations; it determines the *initial bottleneck buffers* $a^{(L)} < \ldots < a^{(1)}$. $a^{(1)}$ is the buffer index of the first buffer in the bottleneck machine (i^*). $a^{(2)}$ is the first index of the first buffer of a machine which acts as a bottleneck upstream to $a^{(1)}$, and so on until $a^{(L)} = 1$. The fluid solution is built such that all of the buffers that are down stream to buffer $a^{(1)}$ keep pace with the bottleneck machine so that it is fully utilized and the system empties at time T^*. We present the algorithm below:

```
**Initialization:**
$B_i^{(0)} := 0$
$M_i^{(0)} := \sum_{k \in C_i} m_k$
$a^{(0)} := K + 1$
**Working Step $n$:**
$K^{(n)} := a^{(n-1)} - 1$
If $K^{(n)} = 0$ then set $L := n - 1$ and stop.
$B_i^{(n)} := 1 - \frac{B_i^{(n-1)}}{M_i^{(n-1)}}$
$M_i^{(n)} := \sum_{k \in C_i \;\; k < a^{(n-1)}} m_k$
$\mu_i^{(n)} := \frac{B_i^{(n)}}{M_i^{(n)}}$
$i^{*(n)} := argmin \; \mu_i^{(n)}$
$a^{(n)} := \min\{k : k \in C_{i^{*(n)}}\}$
Set flows $u_k$ for buffers $k = a^{(n)}, \ldots, K^{(n)}$ to be $\mu_{i^{*(n)}}^{(n)}$.
```

The algorithm categorizes the buffers $k = 1, \ldots, K$ as $k = a^{(L)}, \ldots, K^{(L)}$, $a^{(L-1)}, \ldots, K^{(L-1)}, \ldots\ldots\ldots, a^{(1)}, \ldots, K^{(1)}$. Inspection of the algorithm reveals that it sets $a^{(L)} = 1$, $a^{(1)} = \min\{k : k \in C_{i^*}\}$, $K^{(1)} = K$ and L to be the number of complete iterations that were performed. Within each consecutive set of buffers, $a^{(l)}, \ldots, K^{(l)}$, the algorithm sets constant flow rates for these buffers such that with each iteration, the flow rates are increasing. This implies that $q_k(t) = 0$ for $k = a^{(l)} + 1, \ldots, K^{(l)}$ and that $q_k(t) > 0$ for $k = a^{(L)}, a^{(L-1)}, \ldots, a^{(1)}$.

It is evident that the fluid solution that is calculated by the algorithm maintains the constraints of the fluid model (as presented in Section 2.2). It is also evident that the fluid solution generated by the algorithm empties all of the fluid in the system by T^*.

A schematic of the fluid solution is presented in Figure 5.1. The figure presents the evolution of the amount of upstream fluid $q_k^+(t)$ from time 0 to time T^* [1]. The thick lines above (and to the right of) each label: a^l ($l = L, L-1, \ldots, 1$) corresponds to the

[1] When the problem is taken to be of an infinite multiplicity then the $q_k^+(t)$ values are not defined. Nevertheless, for such problems, the figure shows the evolution of the $q_k(t)$ values.

$q_k^+(t)$ values for buffers $a^{(l)}, \ldots, K^{(l)}$. From these buffers only buffer $a^{(l)}$ has any fluid in it, the rest are empty (and thus their $q_k^+(t)$ are identical).

A cross section of the graph along the vertical red line shows the amount of fluid in each of the buffers at time t. It is seen that only some of the buffers $(a^{(L)}, \ldots, a^{(1)})$ contain a positive amount of fluid. A cross section of the graph along the horizontal green line shows the path of a "single molecule of fluid" or a single job. It is seen that the job waits a positive amount of time in the buffers $a^{(L)}, \ldots, a^{(1)}$ while it spends no time in the other buffers. Specifically, it is seen that each job waits a positive amount of time in buffer $a^{(1)}$. For all buffers $k > a^{(1)}$, the job does not wait.[2]

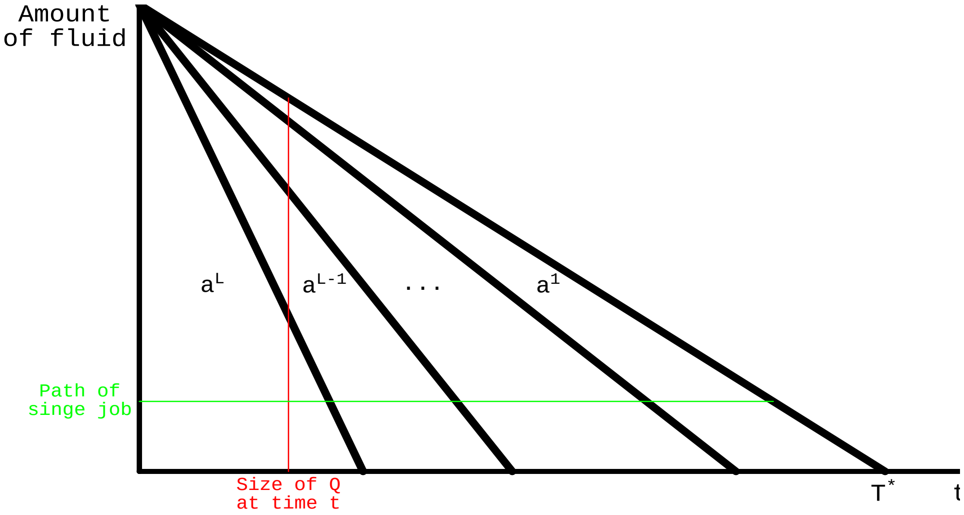


Figure 5.1: The fluid solution of the LBFS algorithm

The Stochastic Model

We assume that the jobs in the re-entrant line have i.i.d. processing times for each operation k with some mean m_k and a finite variance. We define $S_k(t)$ to be the number of jobs that are finished given that machine $\sigma(k)$ works on buffer k for t time units. Thus, $S_k(t)$ is a renewal process (see for example [48]). We define $T_k(t)$ to be the amount of time that machine $\sigma(k)$ works on buffer k during the interval $[0, t)$. As in the job shop model $Q_k(t)$ is the queue for buffer k for $k = 2, \ldots, K$. It is thus evident that $Q_k(t) = S_{k-1}(T_{k-1}(t)) - S_k(T_k(t))$. We assume $Q_1(t)$ is infinite. Note that while

[2] See Weiss [54] for a description of this type of fluid diagram.

being similar to stochastic re-entrant line models discussed in [33], [12] and [13], this model differs in that the initial queue has an infinite amount of input rather than an input according to some arrival process (see Figure 5.2). It is immediately evident that policies such as FBFS may be unstable for some re-entrant lines while policies such as LBFS may be stable (see Section 5.6). Assume now that we process the jobs in the system using a LBFS non-idling policy. Then: $\dot{T}_k(t) = 1$ if and only if $Q_k(t) > 0$, $Q_{k'}(t) = 0$ for $k' > k$ and $k' \in C_{\sigma(k)}$.

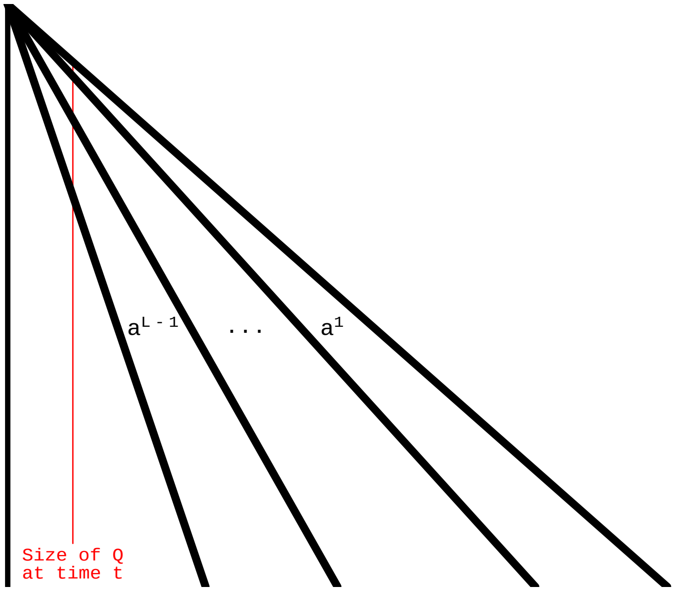


Figure 5.2: The infinite horizon fluid solution

Define $\triangle T_k(t) = T_k(t+1) - T_k(t)$ and $\triangle Q_k(t) = Q_k(t+1) - Q_k(t)$. We conjecture that as $t \to \infty$, we shall have $\triangle T_k(t)$ converge to some steady state distribution with a mean of $m_k u_k$. As a result, $E(S_k(T_k(t+1)) - S_k(T_k(t))) = u_k$. Hence for $k = a^{(L-1)}, \ldots, a^{(1)}$, $\triangle Q_k(t)$ approaches some steady state distribution with a mean of $u_{k-1} - u_k$. While for $k \neq a^{(L-1)}, \ldots, a^{(1)}$, $EQ_k(t+1) = EQ_k(t)$ and $Q_k(t)$ approaches a steady state distribution (it is a stable queue). This leads to the following two conjectures:

Conjecture 1 $\{Q_k(t),\ 2 \leq k \leq K, k \neq a^{(L-1)}, \ldots, a^1, \triangle Q_k(t), k = a^{L-1}, \ldots, a^{(1)}\}$ *converges to a steady state distribution:* $\{Q_k(\infty),\ 2 \leq k \leq K, k \neq a^{(L-1)}, \ldots, a^1, \triangle Q_k(\infty), k = a^{L-1}, \ldots, a^{(1)}\}$ *where* $Q_k(t), Q_k(\infty) \geq 0$ *and* $E(\triangle Q_k(\infty)) = u_{k-1} - u_k > 0$

Conjecture 2 *There exists an R.V., T such that* $Q_k(t) > 0$, $k = a^{(L-1)}, \ldots, a^{(1)}$ *for all* $t > T$.

5.3 The Fluid Imitation Algorithm for Makespan

The fluid imitation algorithm for makespan ($C_{\max}$ FIA) yielded the best results. It showed a $O(1)$ gap for almost all of the problems that we tested. Only for the MT10-bal problem did it feature a gap greater than $O(\log N)$ (but still less than $O(N)$, thus asymptotically optimal)[3]. It was a surprise to find out that the $O(1)$ gap is independent of the processing time distribution. The use of heavy tailed distributions did not cause an increase in the order of the gap. This contradicted our previous belief that was based on the reasoning behind the Dai and Weiss results (their proof relied on the existence of exponential moments). We now detail some of the facts regarding the operation of the algorithm along side an explanation of the empirical results.

Notice that maximization of the lag function presented in Equation 2.10 is exactly like the maximization of this ratio:

$$\frac{Q_k^+(t)}{q_k^+(t)} \tag{5.1}$$

Substituting the fluid solution from Equation 2.8 and noticing that the $(1 - \frac{t}{T^*})$ term is identical for all buffers at a given time t, we see that maximization of the lag is equivalent to maximization of the following[4]:

$$\frac{Q_{(r,o)}^+(t)}{N_r} \tag{5.2}$$

Thus the $C_{\max}$ FIA selects the buffer (r, o) that maximizes 5.2. We shall now discuss how this simple rule applies to various problem types. We start off with re-entrant line problems then continue on to $R \ll J$ identical problems and finally discuss $R \ll J$ proportional problems.

In a re-entrant line problem, $R = 1$ and $N_1 = N$. For such problems the $C_{\max}$ FIA acts exactly like a "schedule buffer with most upstream jobs" rule. At every scheduling epoch, the amount of upstream jobs for each of the non-empty buffers is examined and the next job from the buffer with the most upstream jobs is selected. It is evident that $Q_{(1,o_1)}^+ \leq Q_{(1,o_2)}^+$ for $o_1 < o_2$. At scheduling epochs where both $(1, o_1)$ and $(1, o_2)$ are considered, the inequality is always strong because in order for buffer $(1, o_2)$ to be considered, $Q_{(r,o_2)}(t)$ must be positive. Thus for the re-entrant problem, $C_{\max}$ FIA is the LBFS non-idling rule. Indeed it is seen that the gap for the LBFS simulation run that was performed on an MT10-rline problem was $O(1)$ (as were the results for $C_{\max}$ FIA)[5].

[3] We discuss the MT10-bal problem in the next Chapter.

[4] Notice that we are alternating between the k convention for buffers and the (r, o) convention.

[5] Comparison of the graphs in Appendix Section A.1.1 for the MT10-rline problem using both $C_{\max}$

Conjectures 1 and 2 explain the $O(1)$ behavior of $C_{\max}$ FIA for re-entrant lines as follows: during the execution of the job shop, machines $\sigma(a^{(L)}), \ldots, \sigma(a^{(1)})$ starve only during the initial phase of the job shop (remember that $\sigma(a^{(1)}$ is the bottleneck machine). By Conjecture 1 it is evident that the run-out of the job shop is bounded. This explains the $O(1)$ results for the MT10-rline problems.

For the $R \ll J$ identical case all N_r values are equal. Thus the $C_{\max}$ FIA again acts like a "schedule buffer with most upstream jobs" rule, just like it did for the simpler re-entrant line case. The difference here is that the rule is no longer a LBFS buffer priority dispatching rule. Denote the set of all non empty buffers of scheduleable machine i and time t by $C_i(t)$. Now partition $C_i(t)$ to $C_i^1(t), \ldots, C_i^R(t)$ where $C_i^r(t)$ are the non empty buffers that belong to route r. When comparing the lag of all of the buffers within each $C_i^r(t)$, the $C_{\max}$ FIA uses LBFS. But when comparing the lag of buffers that belong to different $C_i^r(t)$ groups, the rule looks at the amount of upstream jobs, a time dependent state. This is different from the LBFS buffer priority dispatching rule where two buffers that have an equal o are assigned an arbitrary priority based on their route number (r) and this priority is fixed throughout the evolution of the job shop run.

For the proportional case, the rule is "normalized" to the number of jobs in each route in the same manner that the PRBSR normalizes the RBSR (see Section 5.4).

It is thus evident that while the $C_{\max}$ FIA is based on non-trivial concept, a fluid approximation, the scheduling rule is actually quite simple. It should be noted that many of the dispatching rules that have been studied in the literature are rules that are based on the local condition at the machine at question. The $C_{\max}$ FIA is different. While simple to implement, it is an on-line rule that is based on a global condition (it looks at $Q_k^+(t)$ and not $Q_k(t)$).

5.4 Random Buffer Scheduling

We found the RBSR to be asymptotically optimal for the $R \ll J$ identical case! It featured a gap of $O(\log N)$. For the $R \ll J$ proportional case we found it to be non-optimal. For this case, proportional selection of the buffers (PRBSR) achieved asymptotically optimal schedules. An exception is the SWV13 meaned topology; for problems that use this topology the RBSR rule is $O(1)$ for both the identical and proportional cases[6].

FIA and LBFS shows that while both are $O(1)$, the behavior is slightly different. This is due to the behavior of the $C_{\max}$ FIA during time epochs that are greater than T^* (see Section 2.3). We regard this as an implementation detail.

[6] We have not been able to find a good explanation for the fact that the RBSR is $O(1)$ for the SWV13 meaned topology. This result along with the other experiments that use the RBSR and PRBSR rules shows that the behavior of RBSR and PRBSR is topology related.

These results regarding random buffer scheduling are very important because they imply that there is “no pressing need” for complicated scheduling heuristics. In practice, if a semi-conductor manufacturer wanted to achieve asymptotically optimal scheduling in terms of $C_{\max}$ while minimizing the expenses of machine synchronization and control (staff, communications and computing expenses), she could use the RBSR or PRBSR.

Proportional Buffer Scheduling: PRBSR

The RBSR is not optimal for the $R \ll J$ proportional case (see Figure 5.3 in which the relative gap appears to converge to 0.05 and not to 0 as is expected of an asymptotically optimal scheduling rule). We thus devised the PRBSR. The reasoning is as follows: The $R \ll J$ proportional case features a distinct number of jobs (N_r) on each route r. We may also treat the problem as having N_r identical routes with a single job each. In this case, the problem is identical and not proportional. Assuming that the RBSR rule works for this identical problem, it is immediately seen that it acts like applying the PRBSR to the original problem. While the rate of growth of the gap for the PRBSR problems usually appeared to be higher than $O(\log N)$, it was still asymptotically optimal for all of the experiments (see Figure 5.4).

5.5 Decreasing Starve Times

Several of the results have shown that the mean starve-time decreases as N is increased. How is this possible? We believe that it is due to the fact that for low multiplicities the bottleneck occasionally starves because there are not many jobs traveling in the system. For high multiplicities, there is an abundance of jobs in the system from the start. This sometimes ensures that the bottleneck is almost constantly busy.

5.6 Non-optimal Scheduling Rules: FBFS, LBFS and RJSR

The first buffer first serve (FBFS), last buffer first serve (LBFS) and random job scheduling rules (RJSR) are all non-optimal. The graphs of the simulation experiments for these rules show that the gap grows at an exponential rate when plotted against a multiplicity on a logarithmic scale. This implies that the gap grows linearly in N: $O(N)$. Figure 5.5 shows a graph of the ratio $\mathcal{G}(N)/N$ for one of the FBFS simulation experiments. In the figure, it appears that the ratio converges to about 0.34; it does not converge to 0 as one would expect from an asymptotically optimal rule.

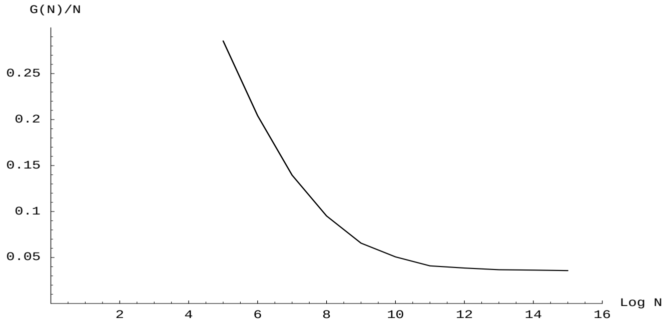


Figure 5.3: The relative gap of C.V. = 0, MT10 using RBSR

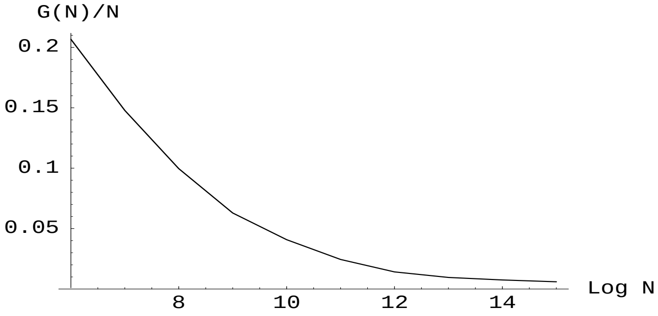


Figure 5.4: The relative gap of C.V. = 0, MT10 using PRBSR

FBFS

We explain why FBFS is non-optimal for the MT10 problem. In this problem the bottleneck machine is machine 4. The first and second operations on all of the routes occurs on machines 1, 2 and 3; remember that we denote these as initial machines. Thus the bottleneck does not perform the first or second operation of any of the routes. Now as long as there are jobs that have still not started their first operation, the initial machines prefer to schedule these un-started jobs (as specified by FBFS). As each operation (r, o), $r = 1, \ldots, R$ is completed on an initial machine, it traverses to a neighboring initial machine, and never to the bottleneck machine. Thus the bottleneck machine is starving for a time that is bounded from below by the time it takes to empty all of the operations from one of the buffers $(r, 1)$, $r = 1, \ldots, 10$. Note that this time is $O(N)$. This analysis is specific to the MT10 meaned topology.

RJSR

The RJSR acts like a randomized version of a "schedule buffer with most jobs" rule. Such a rule acts like a mechanism in which each machine tries to ensure that it has an equal amount of jobs in each of the buffers that surround it. If the controller of each machine assumes that there are arrival processes that behave according to the same probability law for all of the buffers then the "schedule buffer with most jobs" rule seems like a reasonable rule for keeping the machine at stake busy for as long as possible. Since in general, this is not the case for $R \ll J$ problems and since keeping the bottleneck busy is more important than keeping all of the machines busy, the RJSR does not perform well: it does not feed the bottleneck machine so as to keep it constantly busy (as the fluid solution does).

Note that we may use the a similar reasoning to the reasoning that we used above for the FBFS and MT10 to explain why the run we made using RJSR on MT10 exhibits a dominant increasing starve-time and is $O(N)$. Since RJSR acts like a "schedule buffer with most jobs" rule, the initial machines will not schedule (with a high probability) any jobs from non-initial buffers until these buffers have more jobs than the initial buffers. The time that it takes an initial machine to reach such a state is bounded from below by the processing time of 1/10'th of the jobs in the initial buffer. This time is $O(N)$. Thus with a high probability, the bottleneck machine starves for a period of $O(N)$.

LBFS

The LBFS rule was also $O(N)$ for most of the simulation experiments. An exception was the MT10-rline meaned topology for which it was $O(1)$. This was explained in Section 5.3.

5.7 $R \approx J$ Problems

We did not perform an all around investigation of $R \approx J$ problems. We simply lay out the framework for such a simulation investigation. For the sake of preliminary experimentation we tested the RJSR on three classes of problems. All three problems were of the type ($I = 10,\ a = 0$). The processing means of all 10 machines were set to 100 time units. We tested these problems for the C.V.=0.25, C.V.=1.0 and Pareto 2 cases. The resulting graphs are displayed in Appendix Section A.3.

By looking at the results, we make the following conjecture: The starve time decreases with N while the run-out time is constant with respect to N. Thus the gap decreases as N grows.

Our conjecture seems plausible because of the random nature in which the $R \approx J$ job shops are generated and the randomness of the RJSR. The decrease in the starve time follows the same argument as in Section 5.5. The constant run-out time implies that there is a bounded amount of jobs in the system after the bottleneck has finished. This is a plausible result if we assume that the stochastic system that operates is stable in a manner similar to the description in Section 5.2.

For the problems that we generated, there is an equal probability for each of the 10 machines to become the bottleneck machine. This is because the processing means of all of the machines are equal and the routing is completely random. This makes it reasonable to assume that the arrival and service process that each machine encounters has the same probability law. Recall that the RJSR acts like a “noisy” version of the “schedule buffer with most jobs” and that such a rule aims at keeping each machine busy for as long as possible. These facts make it reasonable that RJSR does not allow the machines to starve and keeps the run-out constant.

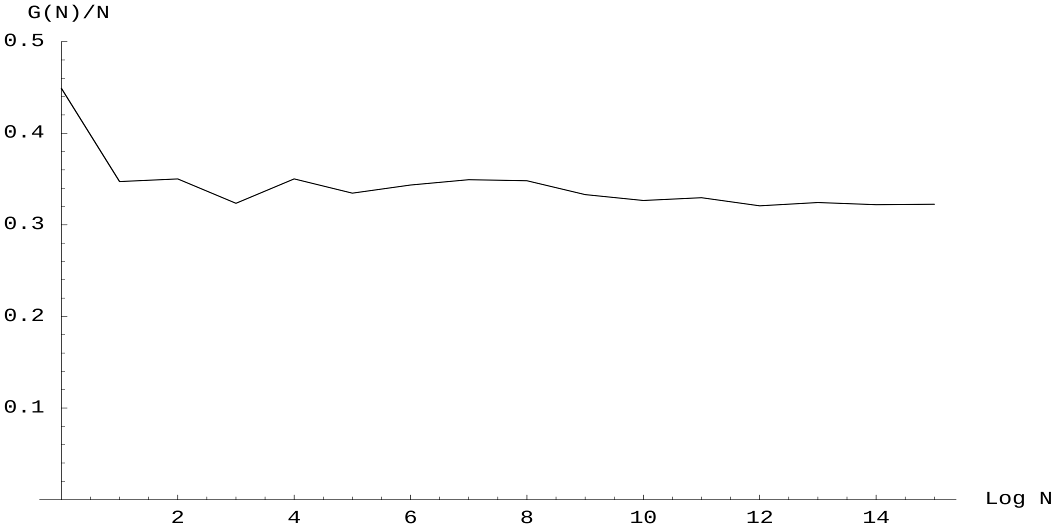


Figure 5.5: The relative gap of Pareto 2, MT10 identical using FBFS

Chapter 6

Multiple Bottleneck Machines

In this chapter we discuss the results that were obtained in the simulation experiments of balanced job shop problems (MT10-bal and MT10-round-bal). We also discuss some steady state models of a 2 machine job shop with 2 opposite routes. In Section 6.1 we discuss the motivation behind job shop problems with multiple bottleneck machines. In Section 6.2 we describe the results that were obtained by the experiments. In Section 6.3 we discuss the 2 machine 2 opposite route problem: 2M2OR. We believe that an understanding of this simple topology is required for understanding of the more general job shop problem. In Section 6.4 we describe the push-pull model by Kopzon and Weiss [32]. This is a steady state Markovian model of the 2M2OR problem using a specific scheduling policy for which the steady state distribution has been derived. In Section 6.5 we describe an infinite horizon model for the 2M2OR problem that is based on the $C_{\max}$ FIA scheduling rule. Finally, in Section 6.6 we discuss the applications and functionality of queueing system models that use infinite virtual queues.

6.1 The Motivation and the Questions

In a production process, one may often wish to utilize all of the machines to their full potential. An un-utilized machine is often conceived as wasted capital. In the job shop formulation, the bottleneck machine is utilized nearly all of the time when scheduling under an asymptotically optimal policy. Non-bottleneck machines are not. We thus wish to set up the job shop such that the meaned topology defines more than one bottleneck machine while still achieving asymptotically optimal scheduling. Preferably, we would like to have all of the machines set up as bottleneck machines.

Unfortunately, previous results regarding asymptotically optimal job shop scheduling have hinted that it is difficult to achieve asymptotically optimal scheduling when there is more than a single bottleneck. The results regarding the Dai-Weiss fluid heuristic (covered in Section 2.4.2) are based on the assumption of a single bottleneck machine. We are thus interested in investigating how job shop problems with multiple bottleneck machines are scheduled by the $C_{\max}$ FIA.

It is important to make the following distinction: For C.V.=0 problems, the processing times of all of the bottleneck machines are equal; while for stochastic problems, only the mean processing times are equal. For such problems, the total processing times of each of the machines are approximately normal random variables with an expectation that equals $N\sum_{(r,o)\in C_i} m_{(r,o)}$ and a standard deviation that is $O(\sqrt{N})$ (because of the CLT). The machine lower bound is the maximum of these I R.V.'s and its expectation is $N\sum_{(r,o)\in C_i} m_{(r,o)} + O(\sqrt{N})$ (see for example David [15]).

6.2 Simulation Experiment Results

As reported in Chapter 4, we conducted experiments for two balanced topologies: MT10-bal and MT10-round-bal. Our results hint that $C_{\max}$ FIA is asymptotically optimal for both of these cases regardless of the processing time distribution that is used. These results were surprising with regard to the assumptions that were made by Dai and Weiss in-order to obtain the result that the gap is $O(\log N)$ (see Section 5.1).

While the simulation experiments of both the MT10-bal and MT10-round-bal problems generated asymptotically optimal schedules, there is a big difference between the two. For the MT10-bal problem, the gap appears to be increasing at a rate that is faster than logarithmic and for the MT10-round-bal problems the gap is observed to be $O(1)$.

In-order to verify that the MT10-bal simulation experiments are asymptotically optimal ($o(N)$) (even though they showed an increasing gap), we examine graphs of their relative gap in Figures 6.1, 6.2 and 6.3 and observe that it approaches 0 as N increases.

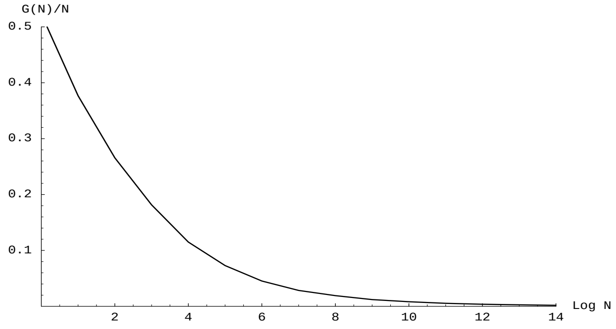


Figure 6.1: The relative gap of C.V.=0.25, MT10-bal, identical using $C_{\max}$ FIA

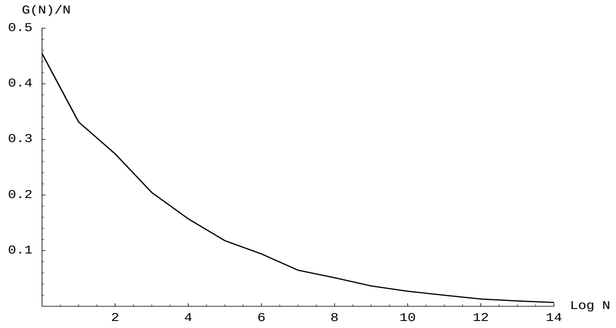


Figure 6.2: The relative gap of C.V.=1.0, MT10-bal, identical using $C_{\max}$ FIA

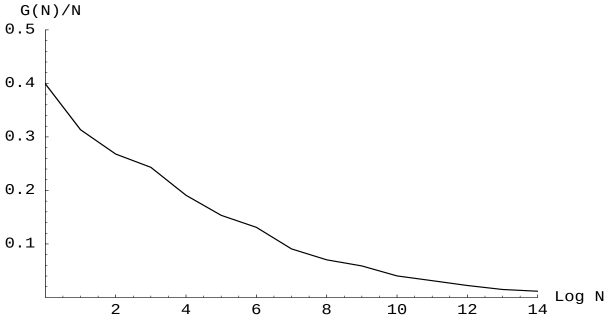


Figure 6.3: The relative gap of Pareto 2, MT10-bal, identical using $C_{\max}$ FIA

6.3 The 2 Machine 2 Opposite Routes Problem

We shall now discuss high volume $R \ll J$ job shop problems with 2 machines and 2 opposite routes using various mean processing time configurations. We call this problem 2M2OR (see Figure 6.4)[1]. We believe that an understanding of this simple job shop problem, may yield a better understanding of the dynamics of complex job shop problems such as MT10-bal and MT10-round-bal.

> *Job shop topology 2M2OR* : $I = 2$, $R = 2$, $K_1 = K_2 = 2$, $\sigma(1,1) = 1$, $\sigma(1,2) = 2$, $\sigma(2,1) = 2$, $\sigma(2,2) = 1$. We enumerate the buffers by k = 1,2,3,4: 1=(1,1), 2 = (1,2), 3=(2,1) and 4=(2,2).

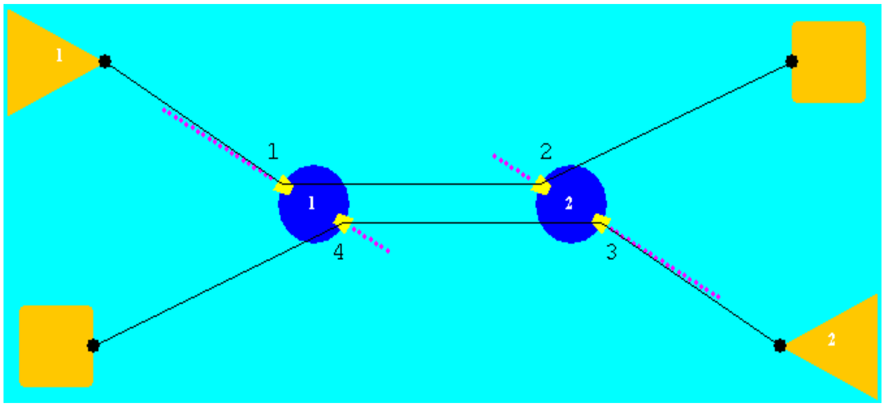


Figure 6.4: The 2M2OR topology as seen in the JSS

We say that machine i is *producing* if it is working on operation $(i,1)$. We say that it is *serving* if it is working on operation $(3-i,2)$. We also say that the queue of machine i is the queue at operation $(3-i,2)$, this is the queue of jobs that are waiting to be "served" at machine i. This terminology is adopted from Kopzon and Weiss [32].

The processing times for the 2M2OR problem may be defined in several ways. We say that the problem is *inherently-unstable* if for each route the mean processing time of the the first operation is less than that of the second operation. We say that it is *inherently-stable* if the opposite occurs. We say that a problem is *mixed* if on one route the first operation is fast (inherently-unstable) and on the other route the first operation is slow (inherently-stable). Note that for a mixed problem the operations of one machine

[1] See Appendix Section C.1 for an explanation of the graphics in the figure.

are slower on both routes, so there can be only one bottleneck, as in Chapter 5, so we do not discuss it here.

We now ask whether it is possible for both machines to be busy all the time while keeping the size of the queues stable. For this to be true the rate of jobs produced on each route must equal the rate of jobs that are served on each route. We denote the rate of jobs produced and served on route i by ν_i, the fraction of time that machine i is serving by α_i and the rate of production of operation i by $\lambda_i = \frac{1}{m_i}$. Thus for a stable operation of the system, the following must hold:

$$\nu_1 = (1-\alpha_1)\lambda_1 = \alpha_2\lambda_2$$
$$\nu_2 = (1-\alpha_2)\lambda_3 = \alpha_1\lambda_4$$

Solving for α_1 and α_2 and substitution in the expressions for ν_1 and ν_2 we have:

$$\nu_1 = \frac{\lambda_1\lambda_2(\lambda_4-\lambda_3)}{\lambda_2\lambda_4-\lambda_1\lambda_2}$$
$$\nu_2 = \frac{\lambda_3\lambda_4(\lambda_2-\lambda_1)}{\lambda_2\lambda_4-\lambda_1\lambda_2}$$

It turns out that for the inherently-stable case there is a stable policy (when the processing times of operations 1 and 3 are exponential). We discuss this case in Section 6.4. We believe that $C_{\max}$ FIA is also a stable policy. We believe that it is stable both for the inherently-stable case and the inherently-unstable case. We discuss this in Section 6.5.

Our belief is supported by simulation experiments that we ran for the 2M2OR with various mean configurations using the $C_{\max}$ FIA (these experiments are not discussed in Chapter 4). The experiments were performed with a maximum log multiplicity of 13 and with sample sizes of 30. C.V.=0, C.V.=1 and Pareto 2 processing times were used. The results of all of the experiments showed a gap of $O(1)$. We thus have reason to believe that a steady-state model that explains how the $C_{\max}$ FIA operates on the 2M2OR exists.

Assuming that our conjecture regarding stability of $C_{\max}$ FIA on the 2M2OR problem is true, we believe that the problem may also perform well in the proportional case (when there are N_1 jobs on route 1 and N_2 jobs on route 2). Note that both machines will not starve until $T_s = \sup\{t : Q_{(r,1)}(t) > 0 \textit{ for both } r = 1 \textit{ and } r = 2\}$. If $\frac{N_1}{N_2} = \frac{\nu_1}{\nu_2}$, then $ET_1 = ET_2$ where $T_i = \sum_{(r,o)\in C_i}\sum_{j:\rho(j)=r} X_{(r,o)}(j)$, so both machines are bottlenecks. In this case we believe that T_s is close to both T_1 and T_2 and thus expect to achieve job shop scheduling with a gap that is $O(1)$. If $\frac{N_1}{N_2} \neq \frac{\nu_1}{\nu_2}$, then only one of the machines is a bottleneck so we should have a gap of $O(1)$ is in Chapter 5.

6.4 The Push-Pull Model

In [32], Kopzon and Weiss introduce a scheduling policy for the 2M2OR problem when there is an infinite supply of work at both routes. They show that under a certain policy a steady state equilibrium exists under the assumption of exponential processing times and *inherent-stability* of the problem. (We say that the 2M2OR problems is inherently-stable if for each route the mean processing time of the the first operation is greater than that of the second operation). Their policy is called the *push-pull* policy.

The push-pull model uses the concept of a "stored job". This means that a buffer may perform an operation, complete it and store the job before sending it to the next operation on the route. The policy works in the following manner:

At every scheduling epoch and for every scheduleable machine i: If the queue of machine i is empty (there are no jobs for "serving") then send the "stored job" to the other machine. If the queue is not empty, service the next job from this queue.

It is shown that using this policy, after an initial period, at any time exactly one of the two queues is not empty. The state space is described using the following states: (a,i), (A,i), (b,i) and (B,i). In states of the type a or b, both machines are producing and there are i jobs in the queue of machine 1 or machine 2 respectively. In states A and B, there is one machine serving and one producing, with i jobs in the queue of the serving machine (1 and 2 respectively)[2].

In [32], the authors obtain the steady state probability distribution of the push-pull model. The authors first use Markov balance equations to obtain the steady state for the exponential case. The authors then continue and use the theory of M/G/1 queue with vacations to solve for the case where the "production" is exponential while the "service" is distributed according to some general distribution. The authors also generalize the system to an $I > 2$ machine system with random routes of length 2.

The results by Kopzon and Weiss are important because they show that a system that is operating with a machine utilization of 100% is stable. See Section 6.6 for a discussion.

[2] The push-pull model may also be described such that the "stored job" feature is removed. This can be achieved by slightly modifying the state-space and the scheduling rule.

6.5 An Infinite Horizon Model of 2M2OR Using $C_{\max}$ FIA

Inspired by the empirical results that imply that $C_{\max}$ FIA is stable and by the push-pull model, we are interested in formulating a queueing model (for $0 < t < \infty$) of $C_{\max}$ FIA for the 2M2OR.

We create the model in the following manner: Take the 2M2OR problem and add two counters, $c_1(t)$ and $c_2(t)$ to buffers $(1,1)$ and $(2,1)$ respectively. Each counter is initialized at time 0 to a value of 0. It is then incremented by 1 whenever a job leaves its buffer (whenever an operation is complete). Thus for any N and t:

$$N - c_1(t) = Q_{(1,1)}(t)$$

$$N - c_2(t) = Q_{(2,1)}(t)$$

$C_{\max}$ FIA operates by scheduling the buffer with most upstream jobs. A $C_{\max}$ FIA implementation on 2M2OR would act in the following manner at every scheduling epoch:

For machine 1:
If $Q_{(2,2)}(t) = 0$ or ($Q_{(2,2)}(t) > 0$ and $N - c_1 > Q_{(2,2)} + (N - c_2)$) then schedule (1,1).
If $Q_{(2,2)}(t) > 0$ and $N - c_1 \leq Q_{(2,2)} + (N - c_2)$ then schedule (2,2).
For machine 2 :
If $Q_{(1,2)}(t) = 0$ or ($Q_{(1,2)}(t) > 0$ and $N - c_2 > Q_{(1,2)} + (N - c_1)$) then schedule (2,1).
If $Q_{(1,2)}(t) > 0$ and $N - c_2 \leq Q_{(1,2)} + (N - c_1)$ then schedule (1,2).

Assume that all processing times are exponentially distributed. In this case we have a Markov jump process. We believe that a solution to this system may be found by defining the state of the system as the vector $(X, Y, D)(t)$:

$$(X,\ Y,\ D)(t) = (Q_{(1,2)}(t),\ -Q_{(2,2)}(t),\ c_1(t) - c_2(t))$$

Using this definition, the $C_{\max}$ FIA rule may be phrased as follows:

For machine 1:
If $Y = 0$ or $D < Y < 0$ then schedule (1,1).
If $Y < 0$ and $Y \leq D$ then schedule (2,2).
For machine 2 :
If $X = 0$ or $0 < X < D$ then schedule (2,1).

If $0 < X$ and $X \geq D$ then schedule (1,2).

$(X,\ Y, D)(t)$ defines a dynamic system where X may take on the values $0, 1, 2, \ldots$ Y may take on the values $0, -1, -2, \ldots$ and D may be any integer. It is evident that the system changes at every time instance in which an operation is complete. In each such time instance, the following vector is added to the system (depending on which operation has just completed):

When (1,1) is finished add $(1,\ 0,\ -1)$.
When (1,2) is finished add $(-1,\ 0,\ 0)$.
When (2,1) is finished add $(0,\ -1,\ 1)$.
When (2,2) is finished add $(0,\ 1,\ 0)$.

Based on the simulation results, we believe that this process is stable even when the system is inherently-unstable.

6.6 Discussion of Queueing Models having Infinite Virtual Queues

We believe that the above queueing models are important because they incorporate *infinite virtual queue*s (IVQ). We define an IVQ as a queue with an infinite amount of jobs. In a manufacturing process one may think of an IVQ as a pile of raw materials that is lying besides a machine waiting to be processed. We have seen that IVQ's allow to create a connection between high volume job shops and queueing systems.

Open queueing networks may be viewed as high volume job shop problems where the random input rates are actually machines that are processing jobs that are fed into the machines using an IVQ. Take for example an M/M/1 single server queue. We may think of the server as machine 2, performing operation $(1, 2)$ at rate $\mu = 1/m_{(1,2)}$. We think of the input rate as machine 1 that is being fed by an IVQ and processes the operations at rate $\lambda = 1/m_{(1,1)}$. If we increase the multiplicity N to infinity or to a large value we obtain the M/M/1 model throughout the $[0, T_s)$ period.

Thus, open queuing networks may be viewed as high volume job shop problems where each input rate is substituted by a machine with an IVQ. The reverse, is not as simple, we may not always view a high volume $R \ll J$ job shop problem as an open queuing network. Note that this duality is only possible when the first machine of each route has only one operation ($|C_{\sigma(r,1)}| = 1\ r = 1, \ldots, R$).

We believe that the solution of queueing networks that feature IVQ's may be an important advance. This is because such problems may incorporate the best of both worlds from the scheduling community and the stochastic processes community. Up to now, models of manufacturing systems were either finite source/finite horizon scheduling problems (such as the job shop problem discussed in this study), or they were stochastic networks in which the source is an arrival rate of requests for job processing (such as discussed by Kumar [33]). We believe that neither of these models is realistic when viewing a manufacturing process from a supply side point of view. The finite horizon problem is not realistic simply because the horizon of most factories is not finite. The stochastic arrival rate problems are also not applicable because in many manufacturing situations the goal is to maximize the utility of the capital (have all machines working) and thus production is always at the best achievable maximum.

The above arguments demonstrate why queuing models with IVQ's may be important. An IVQ may be viewed as an infinite pile of raw materials waiting to be produced or a contract that places raw materials at the entry to the system on demand. The goal of the scheduling rules should be: (1) Utilize all of the machines all of the time by ensuring that enough material is drawn in from the IVQ's such that there is no starvation. (2) Ensure that the amount of materials is kept low (is stable). These are not alternative objectives but rather complementary objectives. A scheduling rule that can maintain both of these objectives is desirable. We believe that the $C_{\max}$ FIA or variants of it such as the model presented in Section 6.5 does the job.

Chapter 7

Summary and Future Directions

In this report we summarized a simulation study of on-line scheduling heuristics for $R \ll J$ high volume job shop problems with respect to the makespan objective function. The report described both the software that was created during the course of the study (the JSSP) and the conjectures that were made while analyzing the simulation results. We shall now briefly state what can be further achieved on both of these frontiers.

The Job Shop Simulation Project

The JSSP is now in a stable and robust state (currently version 1.2). Nevertheless, there are many more features and modifications that could be made.

If one was interested in extending the scope of the investigation to a richer problem, the required modifications could easily be made to the software. This includes the transformation of the software to accommodate for DAG shop or open shop problems as well as addition of set-up times, due dates and other features to the basic job shop model. The summary of the design and implementation presented in Chapter 3, along side the software documentation that is supplied with the source code, should be helpful when performing modifications.

There are several features that one may want to add to the Job Shop Simulator (JSS). This front end GUI could be enhanced so that it is more informative to the user regarding the state of the shop. It might also be helpful to enhance the user controlled algorithm so that the JSS becomes a true job shop and/or MCQN exploration tool. Such an enhancement might involve an "undo" operation and the ability to switch between the user controlled algorithm and other on-line algorithms during the execution of a single simulation run. User defined buffer priority algorithms might also be a helpful feature.

In addition, if the JSSP is to be used for an additional simulation study, one might want to enhance the JSS so that it allows for planning and execution of a mass of simulation runs, including recording the runs in the simulation results database (SRDB). Such a feature may greatly improve research times (when compared to using Mathematica as a front end) if it to be implemented in a user friendly manner.

Empirical Results

The results that were recorded in the SRDB, summarized in Chapter 4 and analyzed in Chapter 5 and Chapter 6 have shed a light on high volume job shop scheduling with respect to makespan. We now summarize our findings:

- The $C_{\max}$ FIA schedules job shop problems such that the makespan is equal to the machine lower bound plus a constant that is independent of the number of jobs. This occurs with a high probability and even when the processing times are drawn from heavy tailed distributions.

- The $C_{\max}$ FIA performs as well for the $R \ll J$ proportional case as it does for the $R \ll J$ identical case.

- When there are multiple bottleneck machines, the $C_{\max}$ FIA does not perform as well. In this case the gap is increasing. Yet it is still asymptotically optimal with a high probability. When the problem is set up such that routes start "all around" the problem as opposed to at several specific machines, the balanced job shop problem performs well (a constant gap).

- When applying the LBFS rule on re-entrant line problems, it acts the same as the $C_{\max}$ FIA heuristic. This is due to the fact that both $C_{\max}$ FIA and LBFS essentially act like the "schedule buffer with most upstream jobs" rule.

- Random buffer scheduling rules work extremely well on $R \ll J$ identical problems. While the gap is increasing (and not constant) with the number of jobs, the simulation results have shown that it is only increasing logarithmically with a high probability.

- Unlike the random buffer scheduling rules, the random job scheduling rule is not asymptotically optimal (selecting a buffer at random yields good results while selecting a job at random yields unsatisfactory results).

- 2 machine job shops with 2 opposite routes are asymptotically optimal when scheduled using the $C_{\max}$ FIA. This is independent of the processing means of each of the operations. A steady state model for this problem was formulated in Chapter 6. We believe that it has a stable solution because the simulation results have shown a gap that is constant with respect to the number of jobs.

The above empirical results are yet to be proven. We believe that the key to proving some of these results lies in the relationship between stable MCQN's and asymptotically optimal job shop problems. This is due to the fact that a high volume job shop problem with a constant run-out time implies that the underlying queueing network that is in operation from time 0 to the time at which the bottleneck machine finishes all of it's jobs is stable.

Bibliography

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

[2] Balas, E. (1968) Machine Sequencing via Disjunctive Graphs: an Implicit Enumeration Algorithm. *Operations Research* **17**, p. 941-957.

[3] Banks, J. & Dai, J. G. (1997) Simulation Studies of Multi-class Queueing Networks. *IIE Transactions* **29**, p. 213 -219.

[4] Bertsimas, D. & Gamarnik D. (1999), Asymptotically Optimal Algorithms for Job Shop Scheduling and Packet Routing. *Journal of Algorithms* **33**, p. 296 - 318.

[5] Bertsimas, D. & Gamarnik D. & Sethuraman, J. (1999), From Fluid Relaxations to Practical Algorithms for Job Shop Scheduling: the Holding Cost Objective. (Submitted for publication).

[6] Boudoukh, T. (1999) Algorithms for solving job shop problems with identical and similar jobs, based on fluid approximation (Hebrew with English synopsis). M.Sc. Thesis, Technion, Haifa, Israel.

[7] Boudoukh, T. & Penn, M. & Weiss, G. (2001) Scheduling Job Shops with Some Identical or Similar Jobs. *Journal of Scheduling* **4**, p. 177-199.

[8] Bratley, P. & Fox, B.L. & Schrage, L.E (1983) *A Guide to Simulation, Second Edition.* New York: Springer-Verlag.

[9] Brinkkotter, W. & Brucker, P. (1999) Solving Open Benchmark Problems for the Job Shop Problem. (to appear).

[10] Carlier J. & Pinson, E. (1989) An Algorithm for Solving the Job-shop Problem. *Management Science* **35(2)**, p. 164-176.

[11] Coe, P.S. & Howell, F.W. & Ibbett, R.N. & Williams, L.M. (1998) A Hierarchical Computer Architecture Design and Simulation Environment. *ACM Transactions on Modeling and Computer Simulation* **8(4)**.

[12] Dai, J.G. (1995) On Positive Harris Recurrence of Multi-class Queueing Networks: A Unified approach via Fluid Limit Models. *Annals of Applied Probability* **5**, p. 49-77.

[13] Dai, J.G. & Weiss, G. (1996) Stability and Instability of Fluid Models for Re-Entrant Lines. *Mathematics of Operations Research* **21**, p. 115-134.

[14] Dai, J.G. & Weiss, G. (2000) A Fluid Heuristic for Minimizing Makespan in Job-Shops. *Operations Research*, to appear.

[15] David, H. (1981) *Order Statistics.* New York: Wiley.

[16] Dell'Amico, M. & Trubian, M. (1993) Applying tabu search to the job shop scheduling problem. *Annals of Operations Research* **41**, p. 231-252.

[17] Eckel, B. (2000) *Thinking In JAVA, Second Edition.* Upper Saddle River: Prentice-Hall.

[18] Fowler, M. & Scott, K. (2000) *UML Distilled, Second Edition.* New York: Addison-Wesley.

[19] Gamma, E. & Helm, R. & Johnson, R. & Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading: Addison Wesley.

[20] Garey, M.R & Johnson, D.S. (1979) *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco: Freeman.

[21] Geary, D.M. (1999) *Graphic Java, Volume II: SWING.* Palo Alto: Sun Microsystems.

[22] Glover, F. & Taillard, E. & Werra, D. (1993) A user's guide to tabu search. *Annals of Operations Research* **41**, p. 3-28.

[23] Gosling J. & Joy, B. & Steele, G. & Bracha, G (2000) *The Java Language Specification, Second Edition.* Mountain View: Sun Microsystems.

[24] Hartmann, A. & Schwetman, H. (1988) *Discrete-Event Simulation of Computer and Communication Systems, In Handbook of Simulation, Edited by J. Banks.* New York: John Wiley & Sons.

[25] Helsgaun, K. (2000) Discrete Event Simulation in Java. *DATALOGISKE SKRIFTER (Writings on Computer Science).* Roskilde University.

[26] Hennesy, J.L. & Patterson, D.A (1996) *Computer Architecture, A Quantitative Approach, Second Edition.* San Francisco: Morgan Kaufmann.

[27] Hilton, C. (1998) Manufacturing Operations System Design and Analysis. *Intel Technology Journal.*

[28] Jain, A.S. & Meeran, S. (1998) A State-of-the-art review of Job-shop Scheduling techniques. Technical report, Department of Applied Physics, Electronic and Mechanical Engineering, University of Dundee, Dundee, Scotland.

[29] Jansen, K. & Solis-Oba, R. & Sviridenko, M. (2000) Makespan Minimization in Job Shops: a Linear Time Approximation Scheme. Preprint.

[30] Joins, J.A. & Roberts, S.D. (1988) *Object-Oriented Simulation, In Handbook of Simulation, Edited by Banks, J..* New York: John Wiley & Sons.

[31] Kelton, W.D. & Sadowski, R.P. & Sadowski, D.A. (1998) *Simulation with Arena.* McGraw-Hill: Boston.

[32] Kopzon, A. & Weiss, G. (2001) A Push Pull Queueing System. Preprint.

[33] Kumar P.R. (1995) Scheduling Queueing Networks: Stability, Performance, Analysis and Design. In *Stochastic Networks — IMA Volumes in Mathematics and its Applications*, Vol 71, ed: Kelly F. P. & Williams R.J., Springer-Verlag, New York, p. 21-70.

[34] Kogge, P.M. (1981) *The Architecture of Pipelined Computers.* New York: McGraw-Hill.

[35] Lawler, E.L. & Lenstra, J.K & Rinnoy Kan, A.H.G & Shmoys, D.B. (1993) *Sequencing and Scheduling: Algorithms and Complexity, In Logistics of Production and Inventory, Edited by Graves, S.C. & Rinnoy Kan, A.H.G. & Zipkin, P.H.*, Elsevier Science Publishers: New York.

[36] Lea, D. (1999) *Concurrent Programming in Java, Second Edition.* Mountain View: Sun Microsystems.

[37] Lourenco, H.R.D. (1995) Job-Shop Scheduling: Computational Study of Local Search and Large-Step Optimization Methods. *European Journal of Operational Research*, **83**, p. 347-364.

[38] Manne, A.S. (1960) On the job shop scheduling problem. *Operations Research* **8**, p. 219-223.

[39] Martin, P. & Shmoys, D.B. (1996) A new approach to computing optimal schedules for the job-shop scheduling problem. *International IPCO Conference.* p. 389-403.

[40] Mattfeld, D.C. & Vaessens, R.J.M. (2001) Job Shop Problems in the OR library: `http://mscmga.ms.ic.ac.uk/info.html`

[41] McNab, R. & Howell, F.W. (1996) Using Java for Discrete Event Simulation. *Twelfth UK Computer and Telecommunications Performance Engineering Workshop (UKPEW). University of Edinburgh.* p. 219-228.

[42] Muth, J.F. & Thompson, G.L. (1954) *Industrial Scheduling.* New Jersey: Prentice Hall.

[43] Nowicki, E & Smutnicki, C. (1996) A fast tabu search algorithm for the job shop problem, *Management Science* **42**, p. 797-813.

[44] Pinedo, M. (1995) *Scheduling, Theory, Algorithms and Systems.* New Jersey: Prentice Hall.

[45] Pinedo, M. & Chao, X. (1999) *Operations Scheduling with Applications in Manufacturing and Services.* Boston: Irwin/McGraw-Hill.

[46] Pullan, M.C. (1995) Forms of optimal solutions for separated continuous linear programs. *SIAM J. Control and Optimization* **33**, p. 1952–1977.

[47] Pullan, M.C. (1996) A duality theory for separated continuous linear programs. *SIAM J. Control and Optimization* **34**, p. 931–965.

[48] Ross, S.M. (1983) *Stochastic Processes.* New York: John Wiley & Sons.

[49] Schriber, T.J. & Brunner, D.T. (1988) *How Discrete-Event Simulation Software Works, In Handbook of Simulation, Edited by J. Banks.* New York: John Wiley & Sons.

[50] Sevast'yanov, S.V. (1986) Bounding Algorithm for the Routing Problem with Arbitrary Paths and Alternative Servers. *Kibernetika* **6**, p. 74-79.

[51] Sevast'yanov, S.V. (1994) On some geometric methods in scheduling theory, a survey. *Discrete Applied Mathematics* **55**, p. 59-82.

[52] Storer, R.H & Wu, S.D. & Vaccari R. (1992) New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling. *Management Science* **38(10)**, p. 1495-1509.

[53] Van Laarhoven, P.J.M. & Arts, E.H.L. & Lenstra, J.K. (1992) Job shop scheduling by simulated annealing. *Operations Research* **40**, p. 59-82.

[54] Weiss, G. (1995) On Optimal Draining of Fluid Re-Entrant Lines. In *Stochastic Networks — IMA Volumes in Mathematics and its Applications*, Vol 71, ed: Kelly F. P. & Williams R.J., Springer-Verlag, New York, p. 93-105.

[55] Weiss, G. (1999) Scheduling and Control of Manufacturing Systems - a Fluid Approach. *Proceedings of the 37 Allerton Conference, 21-24 September, 1999, Monticello, Illinois.* p. 557-586.

[56] Weiss, G. (2001) A Simplex Based Algorithm to Solve Separated Continuous Linear Programs. Preprint.

[57] Williamson, D.P. & Hall, L.A. & Hoogeveen, J.A., & Hurkens, C.A.J, & Lenstra, J.K. & Sevast'ynaov, S.V. & Shmoys, D.B. (1997) Short Shop Schedules. *Operations Research* **45**, p. 288-294.

[58] Wolfram, S. (1999) *The Mathematica Book, Fourth Edition.* Wolfram Research: Champaign.

Appendix A

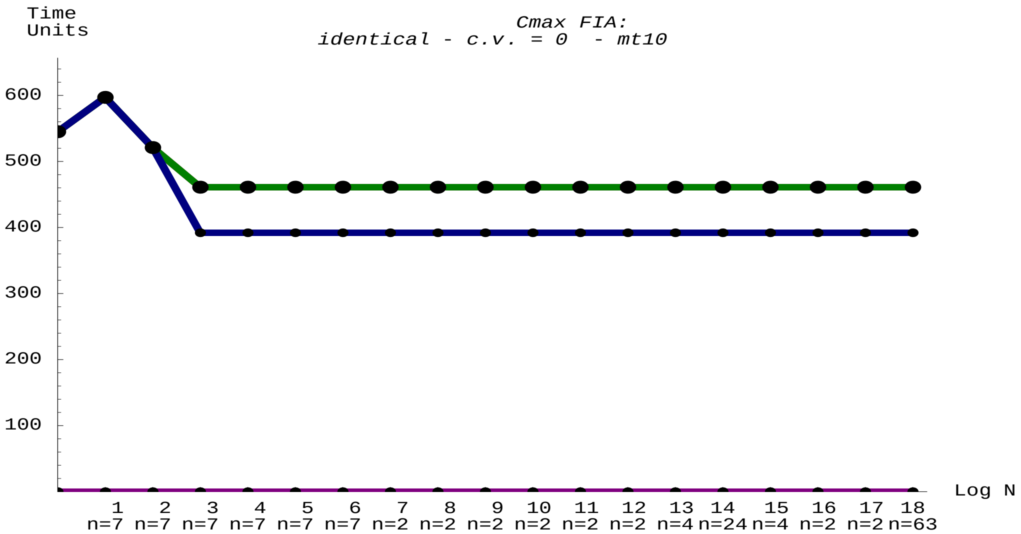
Full Simulation Results

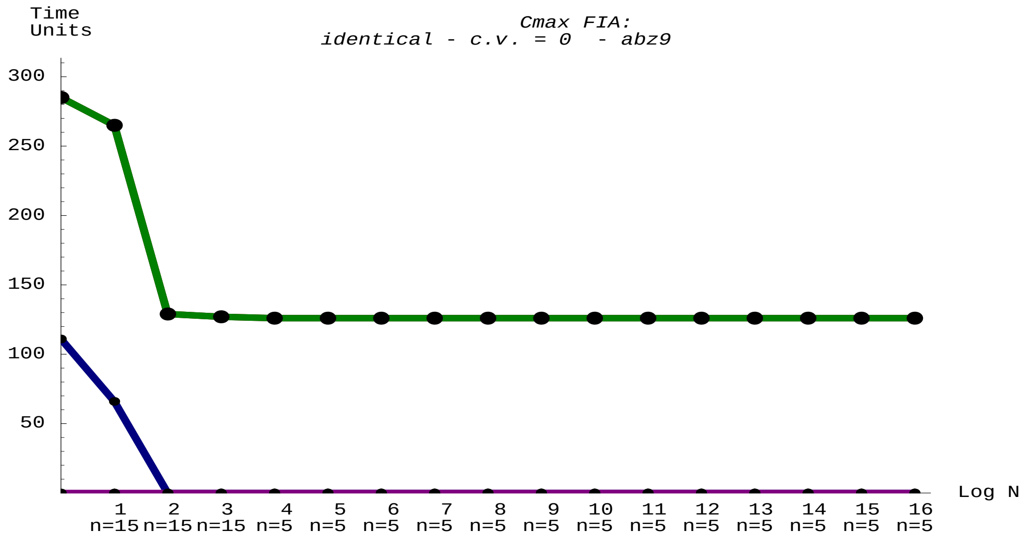
The graphs of the results that were collected are presented below. The contents of the graphs is described in Section 4.4. All of the runs plotted on a single graph belong to the same class of job shop problems and were scheduled using the same algorithm (this is a single *simulation experiment* as defined in Chapter 4).

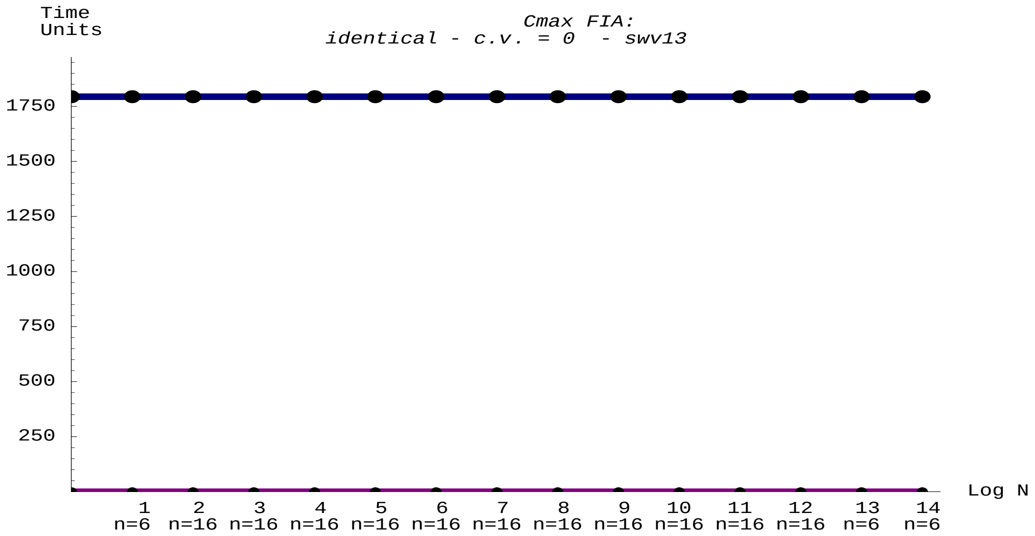
A.1 The $R \ll J$ Identical Case

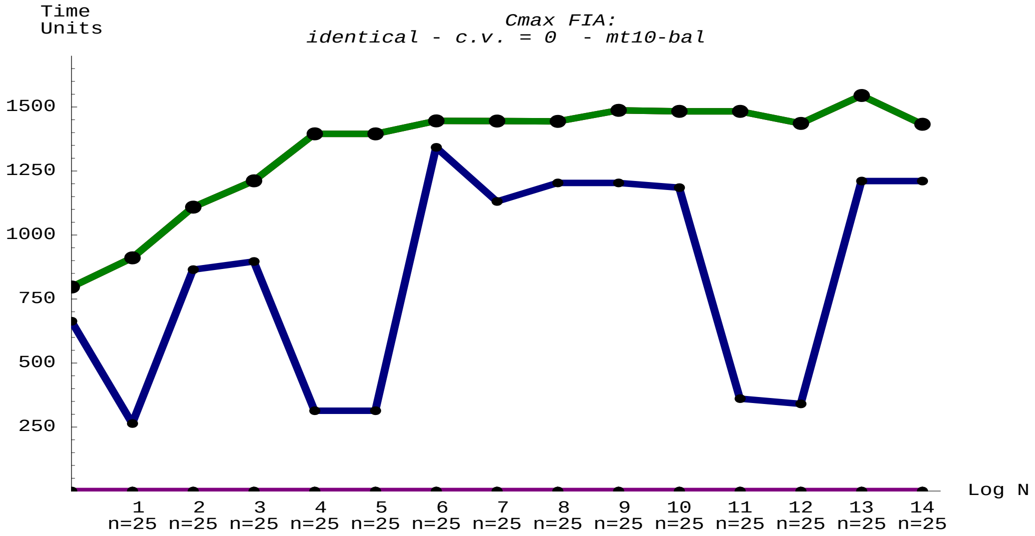
A.1.1 C.V. = 0

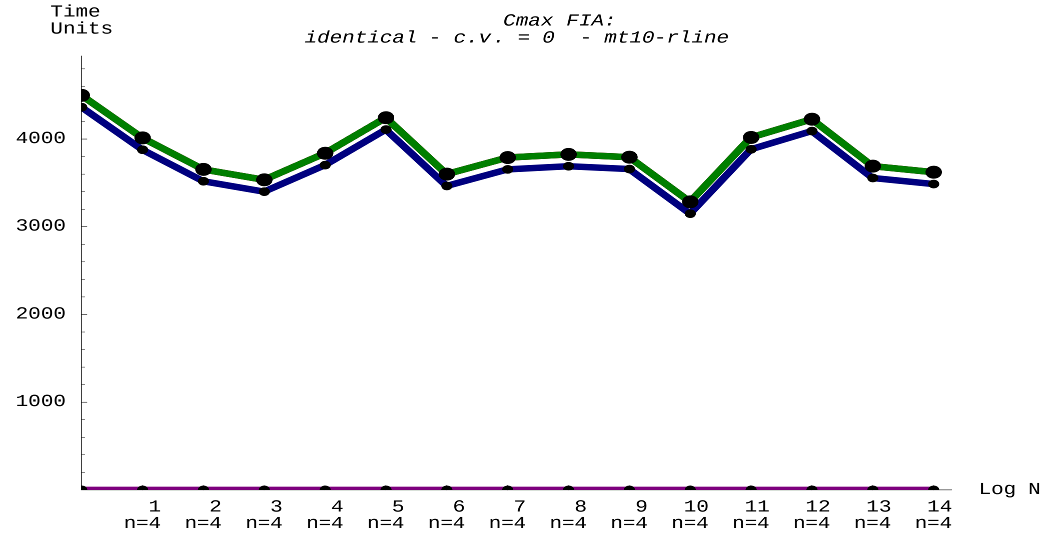
$C_{\max}$ **FIA**



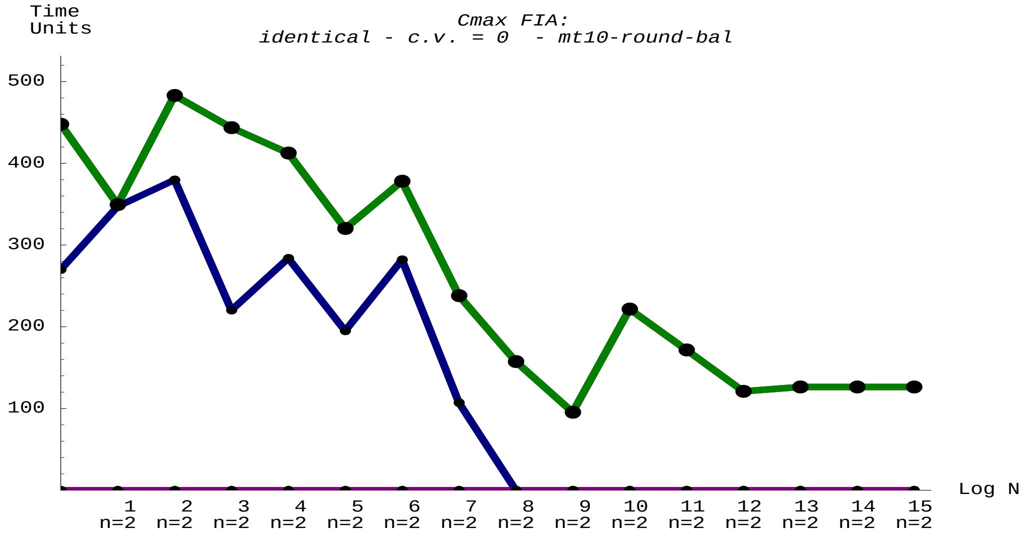

Time
Units
Cmax FIA:
identical - c.v. = 0 - abz9
300
250
200
150
100
50
Log N
1 n=15
2 n=15
3 n=15
4 n=5
5 n=5
6 n=5
7 n=5
8 n=5
9 n=5
10 n=5
11 n=5
12 n=5
13 n=5
14 n=5
15 n=5
16 n=5


Time
Units
Cmax FIA:
identical - c.v. = 0 - swv13
1750
1500
1250
1000
750
500
250
Log N
1 n=6
2 n=16
3 n=16
4 n=16
5 n=16
6 n=16
7 n=16
8 n=16
9 n=16
10 n=16
11 n=16
12 n=16
13 n=6
14 n=6


Time
Units
Cmax FIA:
identical - c.v. = 0 - mt10-bal
1500
1250
1000
750
500
250
Log N
1 n=25
2 n=25
3 n=25
4 n=25
5 n=25
6 n=25
7 n=25
8 n=25
9 n=25
10 n=25
11 n=25
12 n=25
13 n=25
14 n=25

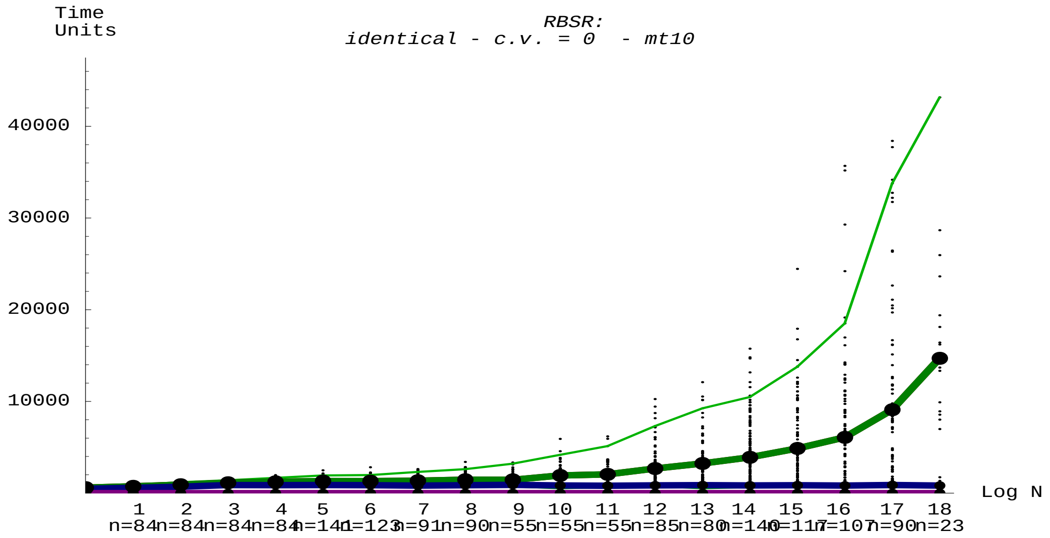


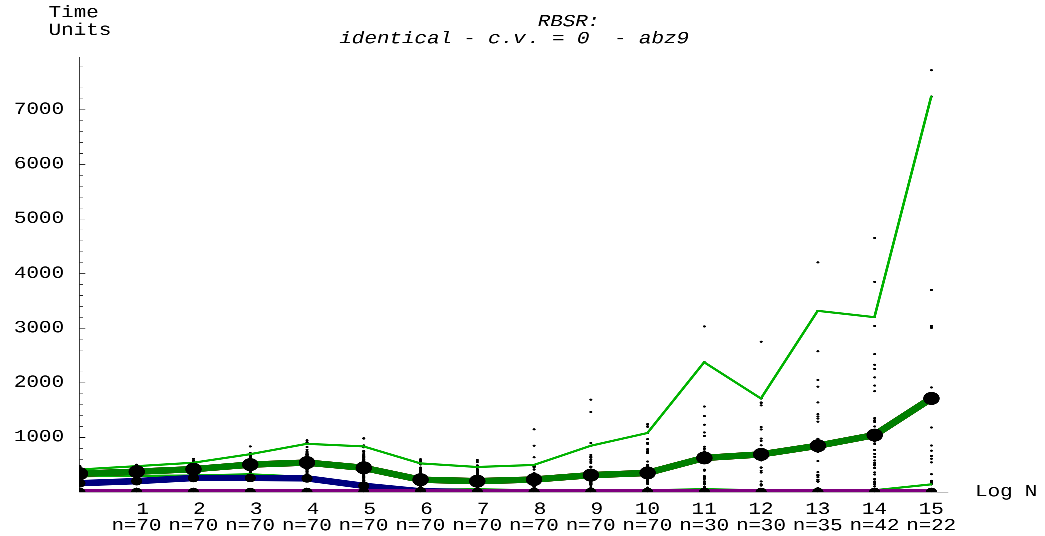
Time
Units
Cmax FIA:
identical - c.v. = 0 - mt10-rline
4000
3000
2000
1000
Log N
1 n=4
2 n=4
3 n=4
4 n=4
5 n=4
6 n=4
7 n=4
8 n=4
9 n=4
10 n=4
11 n=4
12 n=4
13 n=4
14 n=4

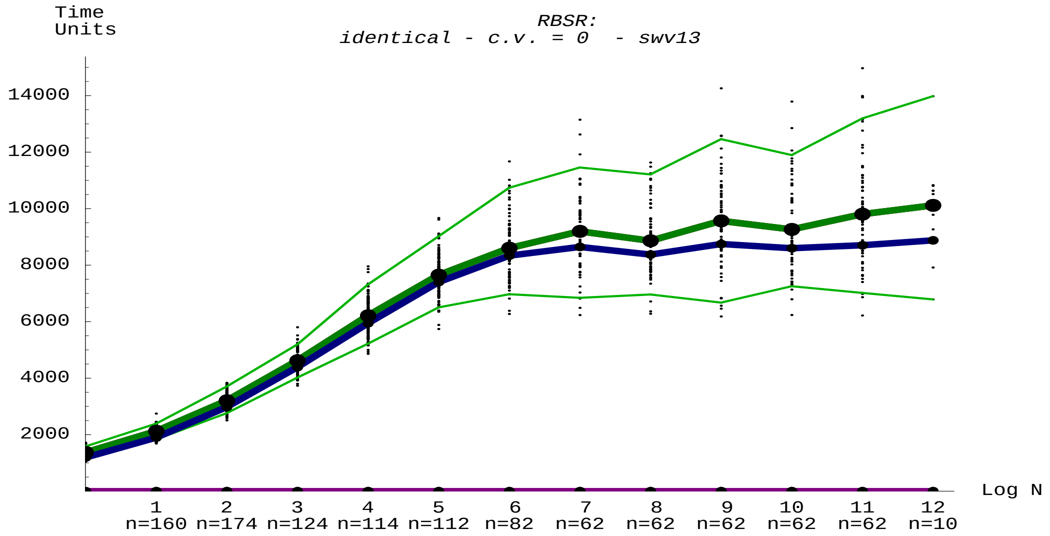


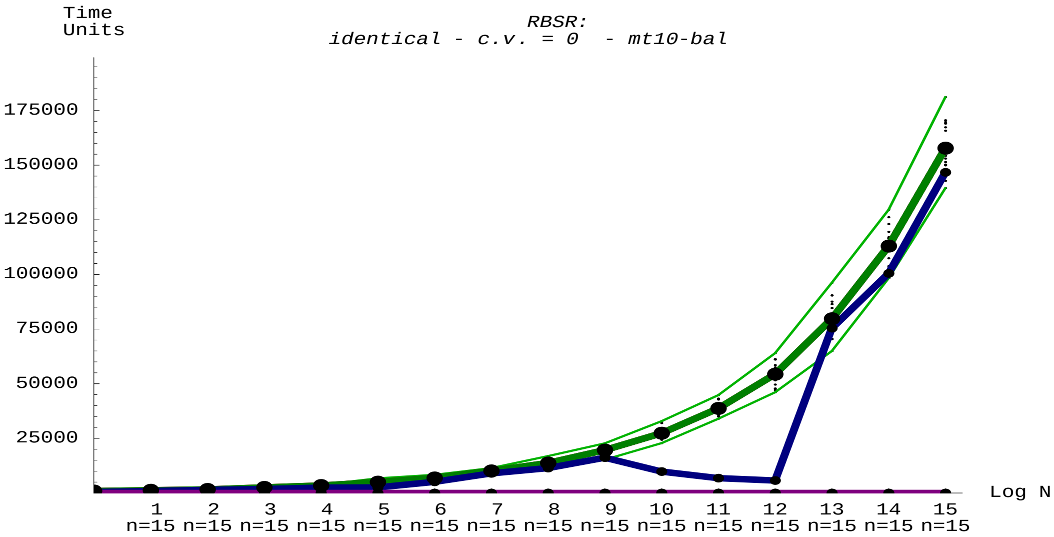
Time
Units
Cmax FIA:
identical - c.v. = 0 - mt10-round-bal
500
400
300
200
100
Log N
1 n=2
2 n=2
3 n=2
4 n=2
5 n=2
6 n=2
7 n=2
8 n=2
9 n=2
10 n=2
11 n=2
12 n=2
13 n=2
14 n=2
15 n=2

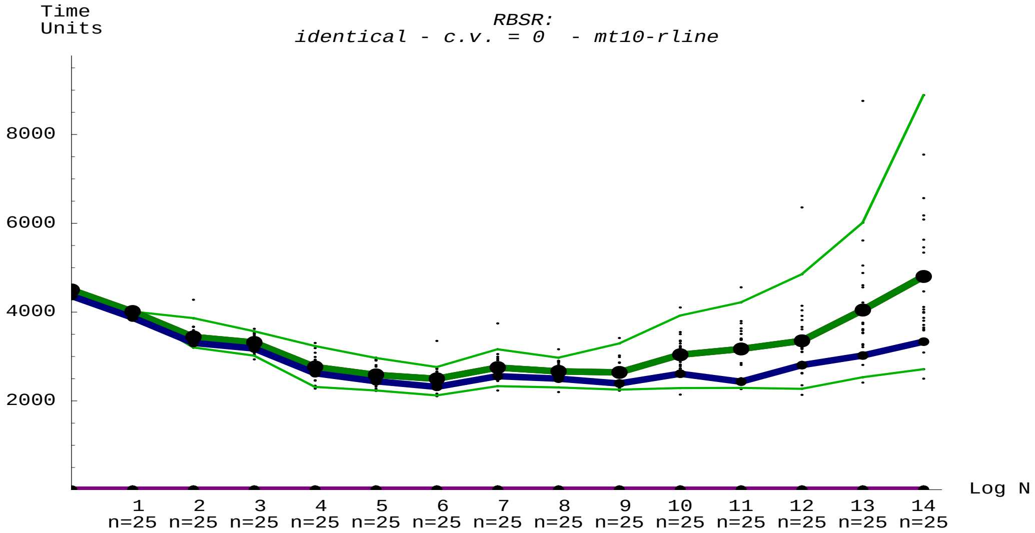
RBSR

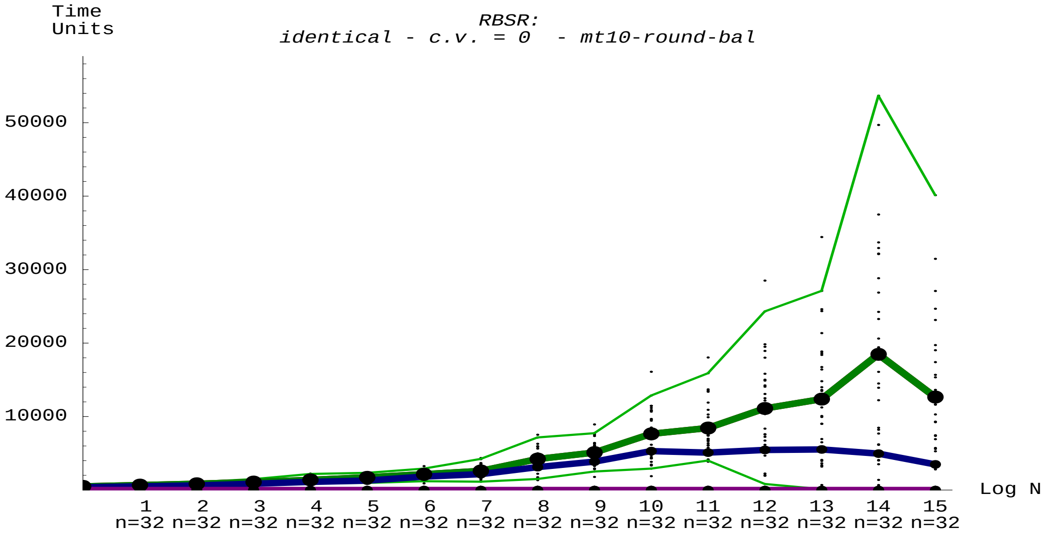

Time
Units
RBSR:
identical - c.v. = 0 - mt10
40000
30000
20000
10000
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
n=84 n=84 n=84 n=84 n=141 n=123 n=91 n=90 n=55 n=55 n=55 n=85 n=80 n=140 n=117 n=107 n=90 n=23


Time
Units
RBSR:
identical - c.v. = 0 - abz9
7000
6000
5000
4000
3000
2000
1000
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
n=70 n=70 n=70 n=70 n=70 n=70 n=70 n=70 n=70 n=70 n=30 n=30 n=35 n=42 n=22

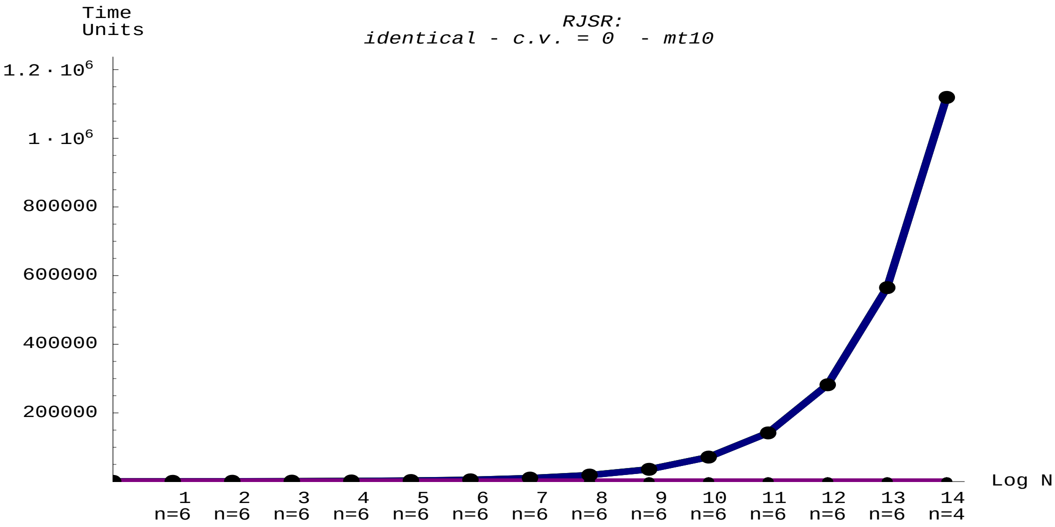

Time
Units
RBSR:
identical - c.v. = 0 - swv13
14000
12000
10000
8000
6000
4000
2000
Log N
1 2 3 4 5 6 7 8 9 10 11 12
n=160 n=174 n=124 n=114 n=112 n=82 n=62 n=62 n=62 n=62 n=62 n=10


Time
Units
RBSR:
identical - c.v. = 0 - mt10-bal
175000
150000
125000
100000
75000
50000
25000
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15

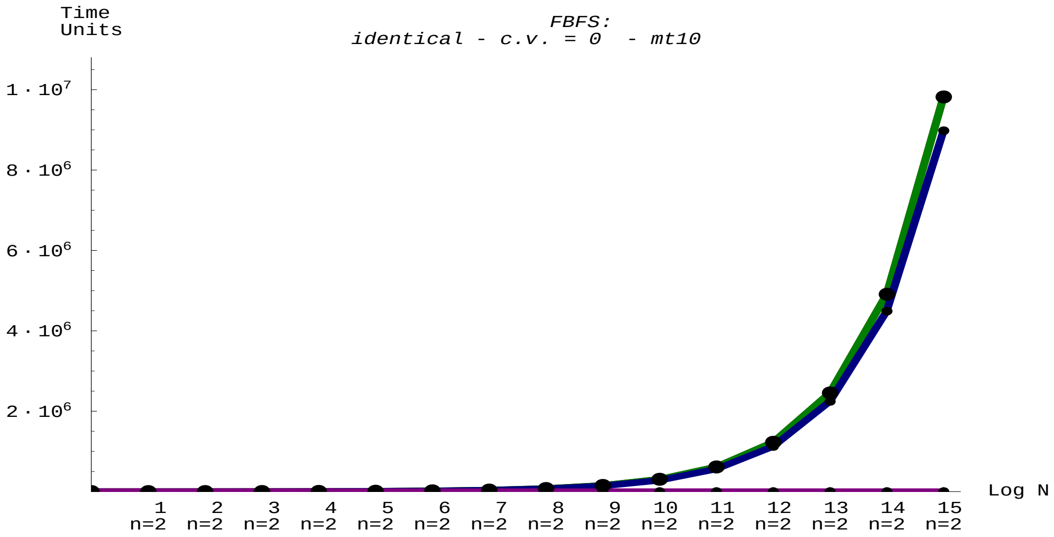

Time
Units
RBSR:
identical - c.v. = 0 - mt10-rline
8000
6000
4000
2000
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14
n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25

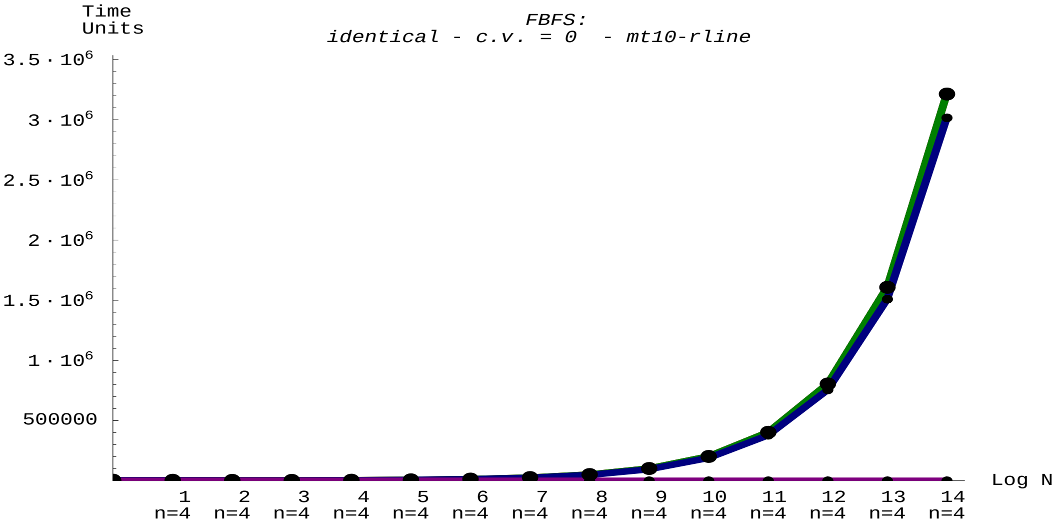

Time
Units
RBSR:
identical - c.v. = 0 - mt10-round-bal
50000
40000
30000
20000
10000
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
n=32 n=32 n=32 n=32 n=32 n=32 n=32 n=32 n=32 n=32 n=32 n=32 n=32 n=32 n=32

RJSR

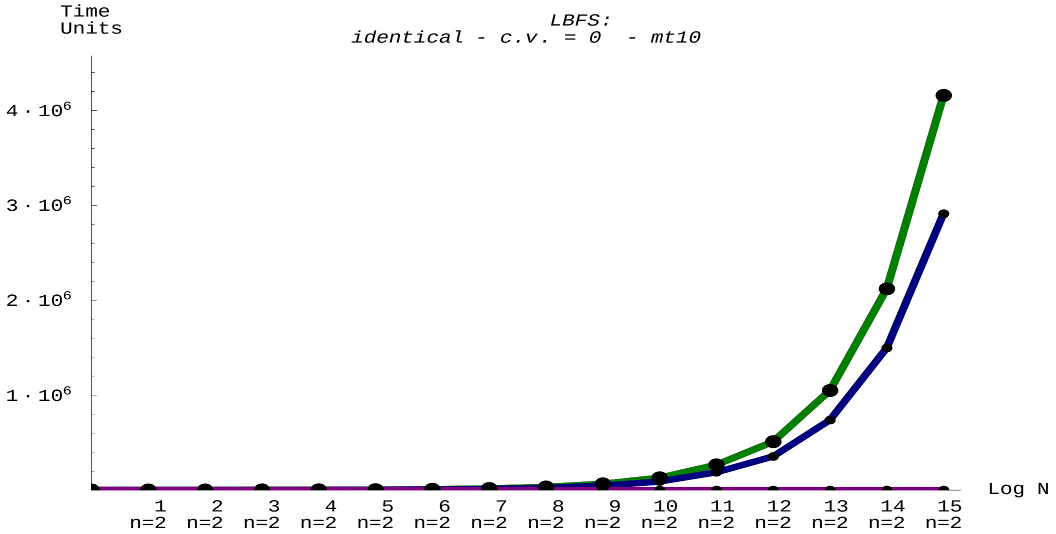

Time
Units
RJSR:
identical - c.v. = 0 - mt10
1.2 · 10^6
1 · 10^6
800000
600000
400000
200000
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14
n=6 n=6 n=6 n=6 n=6 n=6 n=6 n=6 n=6 n=6 n=6 n=6 n=6 n=4

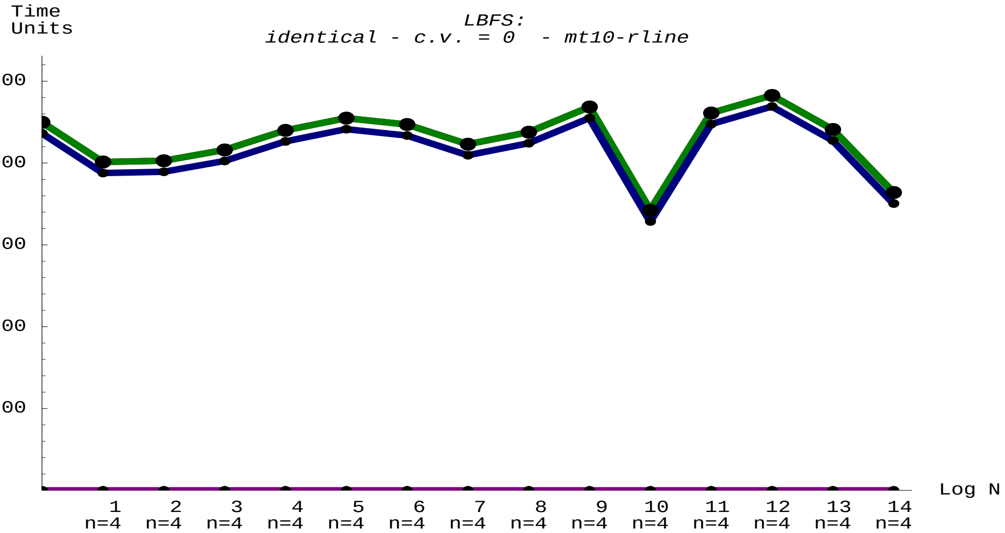
FBFS


Time
Units
FBFS:
identical - c.v. = 0 - mt10
1 · 10^7
8 · 10^6
6 · 10^6
4 · 10^6
2 · 10^6
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
n=2 n=2 n=2 n=2 n=2 n=2 n=2 n=2 n=2 n=2 n=2 n=2 n=2 n=2 n=2


Time
Units
FBFS:
identical - c.v. = 0 - mt10-rline
$3.5 \cdot 10^6$
$3 \cdot 10^6$
$2.5 \cdot 10^6$
$2 \cdot 10^6$
$1.5 \cdot 10^6$
$1 \cdot 10^6$
500000
Log N
1 n=4
2 n=4
3 n=4
4 n=4
5 n=4
6 n=4
7 n=4
8 n=4
9 n=4
10 n=4
11 n=4
12 n=4
13 n=4
14 n=4

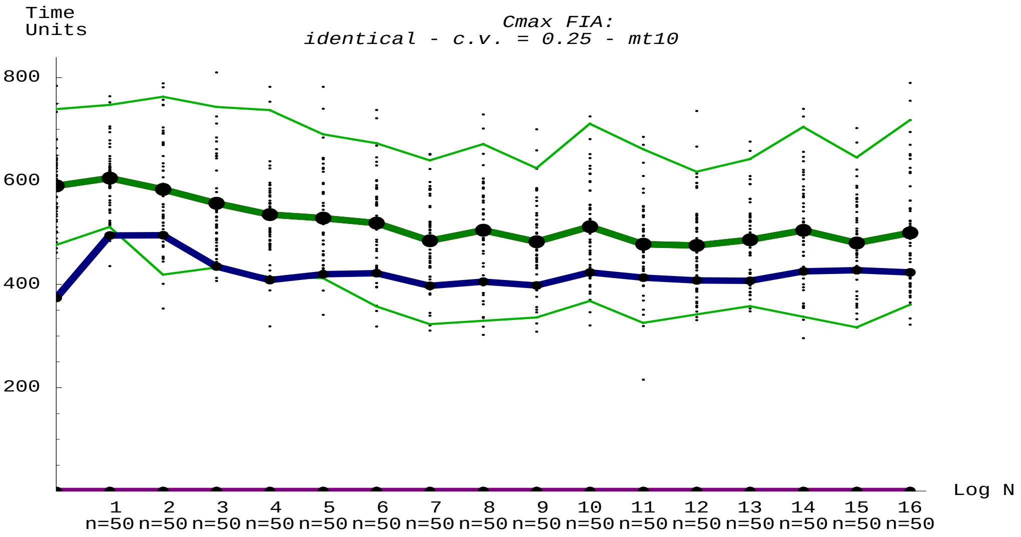
LBFS

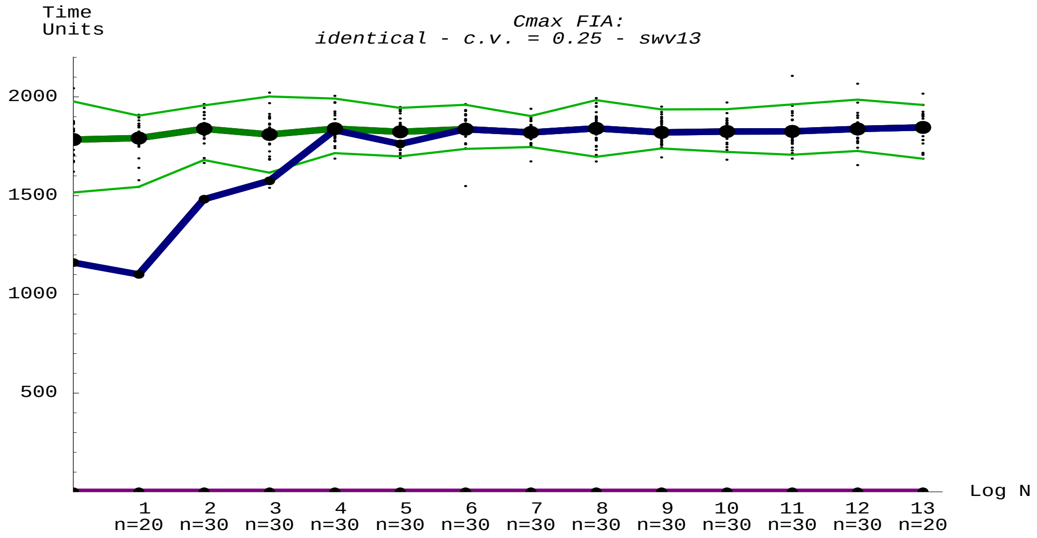

Time
Units
LBFS:
identical - c.v. = 0 - mt10
$4 \cdot 10^6$
$3 \cdot 10^6$
$2 \cdot 10^6$
$1 \cdot 10^6$
Log N
1 n=2
2 n=2
3 n=2
4 n=2
5 n=2
6 n=2
7 n=2
8 n=2
9 n=2
10 n=2
11 n=2
12 n=2
13 n=2
14 n=2
15 n=2

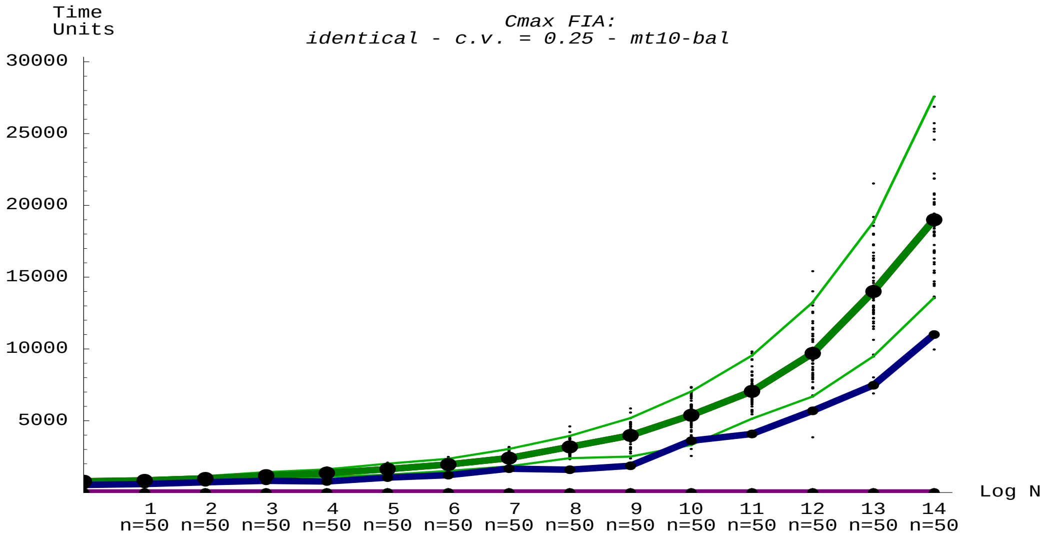


A.1.2 C.V.= 0.25

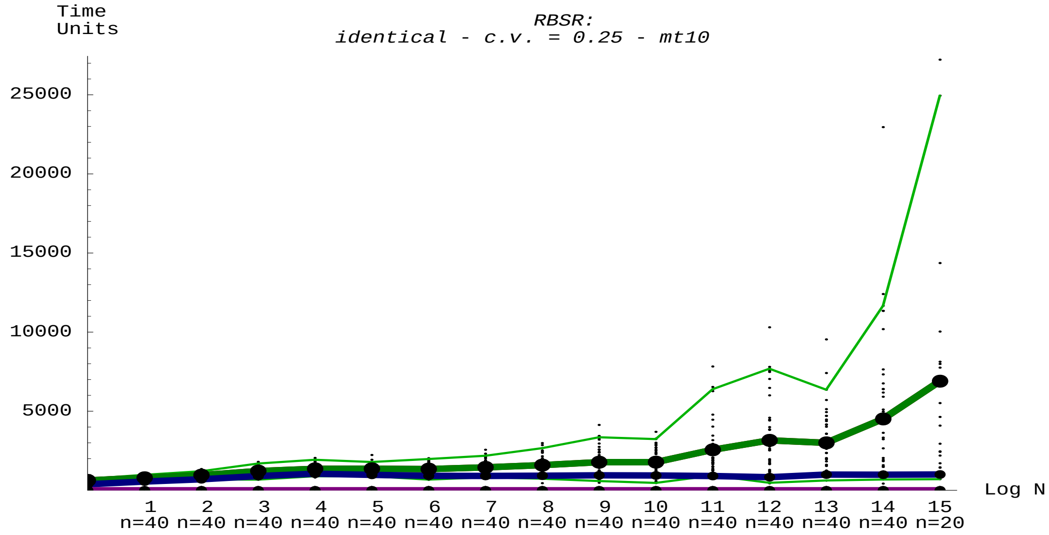
$C_{\max}$ **FIA**

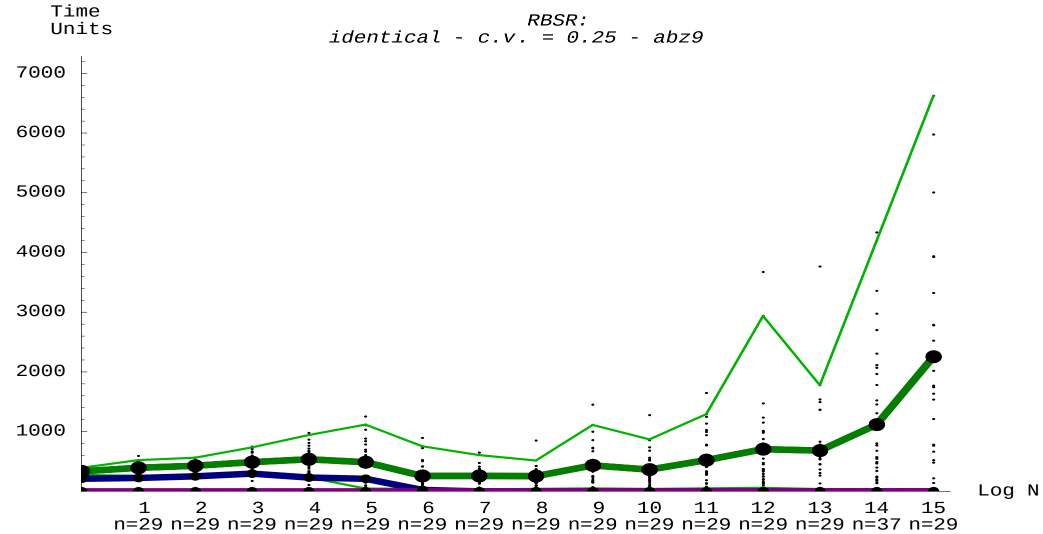


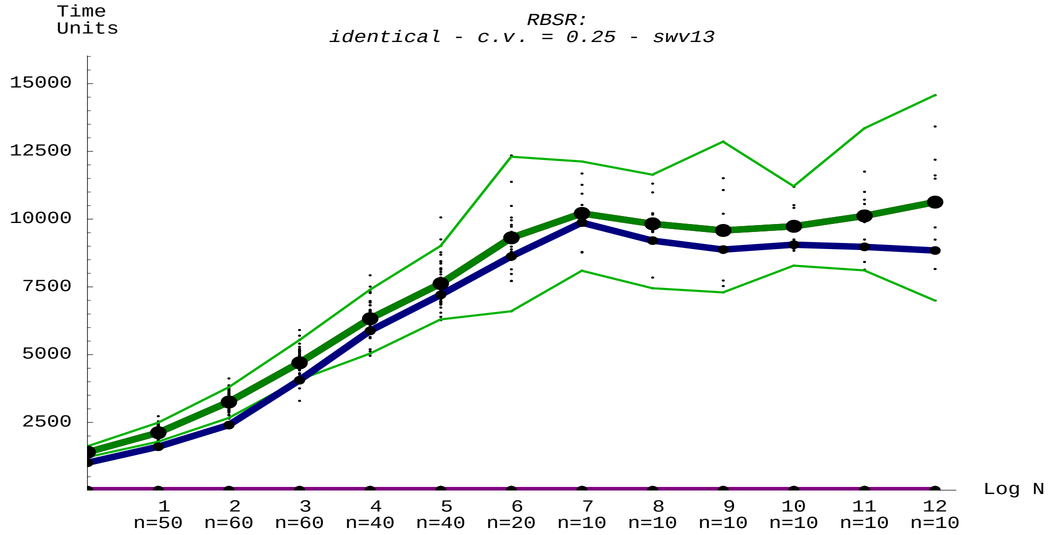

Time
Units
Cmax FIA:
identical - c.v. = 0.25 - swv13
2000
1500
1000
500
Log N
1
n=20
2
n=30
3
n=30
4
n=30
5
n=30
6
n=30
7
n=30
8
n=30
9
n=30
10
n=30
11
n=30
12
n=30
13
n=20

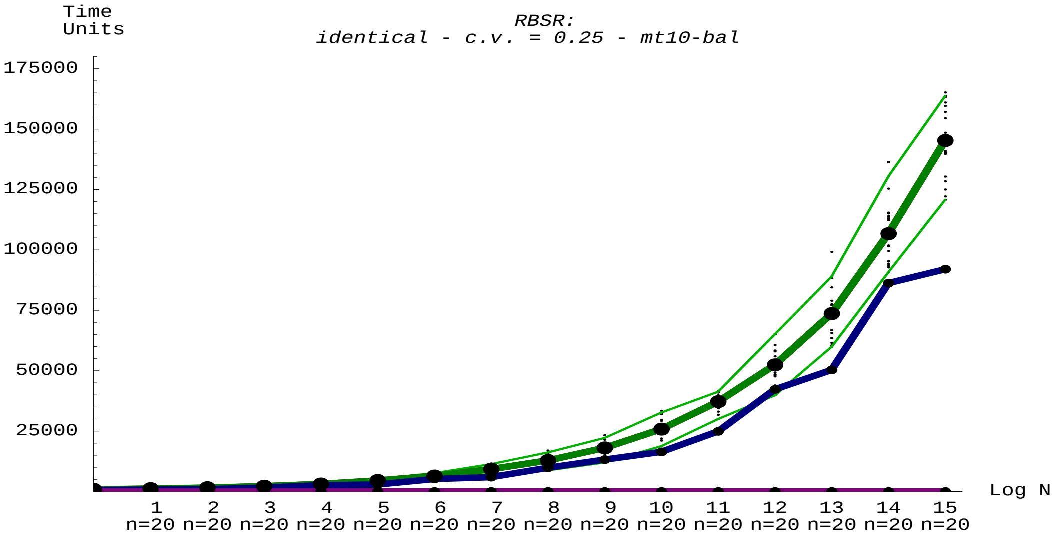

Time
Units
Cmax FIA:
identical - c.v. = 0.25 - mt10-bal
30000
25000
20000
15000
10000
5000
Log N
1
n=50
2
n=50
3
n=50
4
n=50
5
n=50
6
n=50
7
n=50
8
n=50
9
n=50
10
n=50
11
n=50
12
n=50
13
n=50
14
n=50

RBSR

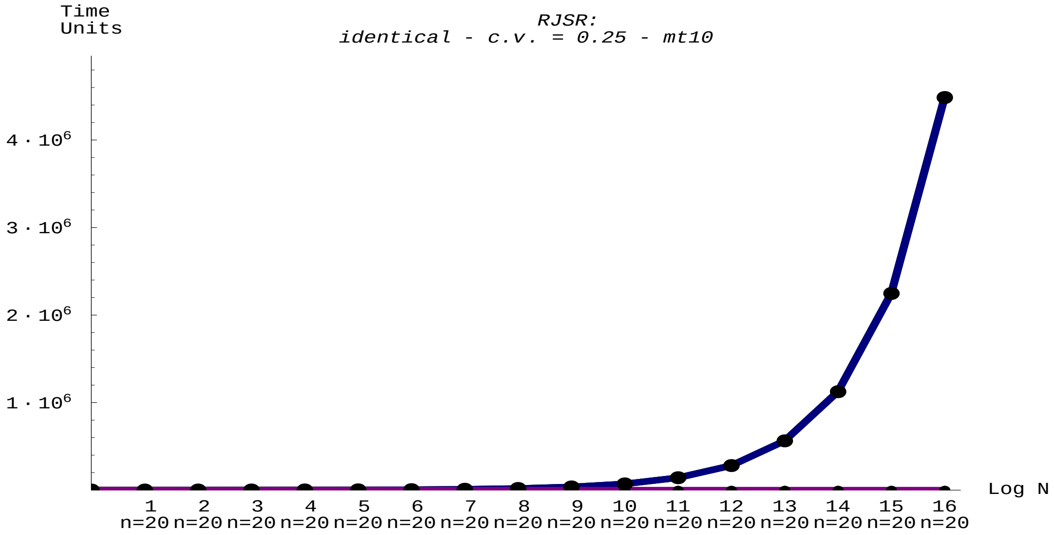

Time
Units
RBSR:
identical - c.v. = 0.25 - mt10
25000
20000
15000
10000
5000
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
n=40 n=40 n=40 n=40 n=40 n=40 n=40 n=40 n=40 n=40 n=40 n=40 n=40 n=40 n=20

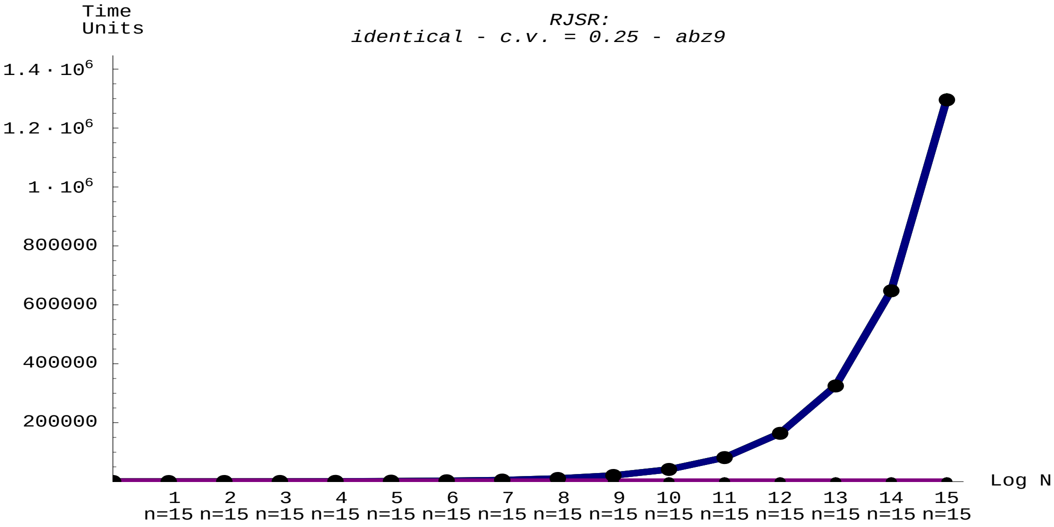

Time
Units
RBSR:
identical - c.v. = 0.25 - abz9
7000
6000
5000
4000
3000
2000
1000
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
n=29 n=29 n=29 n=29 n=29 n=29 n=29 n=29 n=29 n=29 n=29 n=29 n=29 n=37 n=29


Time
Units
RBSR:
identical - c.v. = 0.25 - swv13
15000
12500
10000
7500
5000
2500
Log N
1 2 3 4 5 6 7 8 9 10 11 12
n=50 n=60 n=60 n=40 n=40 n=20 n=10 n=10 n=10 n=10 n=10 n=10


Time
Units
RBSR:
identical - c.v. = 0.25 - mt10-bal
175000
150000
125000
100000
75000
50000
25000
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20

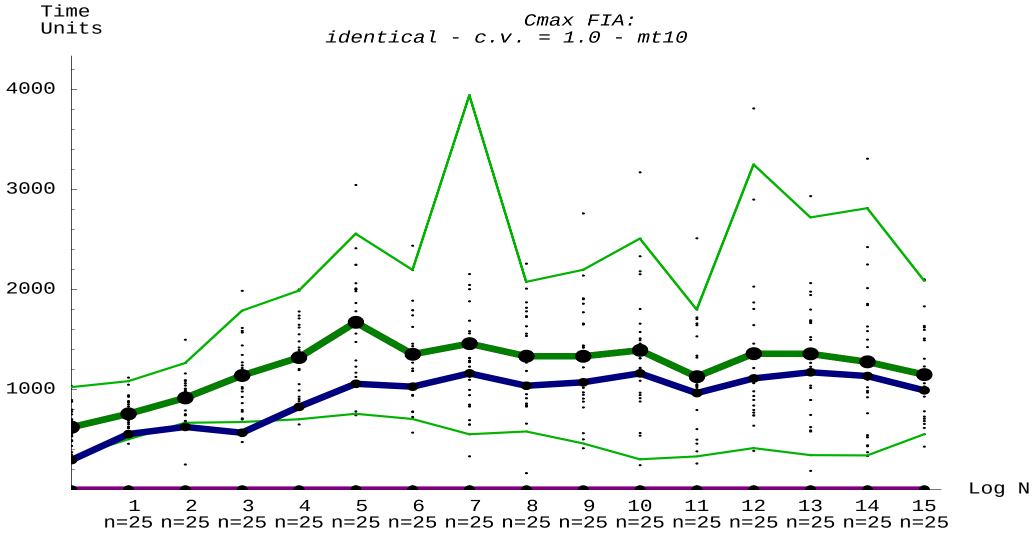
RJSR

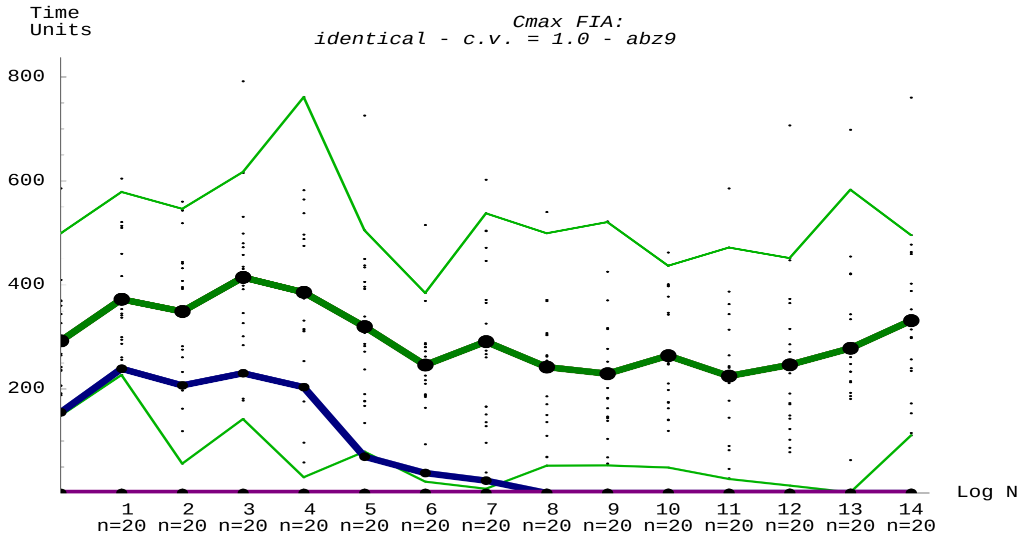

Time
Units
RJSR:
identical - c.v. = 0.25 - mt10
$4 \cdot 10^6$
$3 \cdot 10^6$
$2 \cdot 10^6$
$1 \cdot 10^6$
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20 n=20



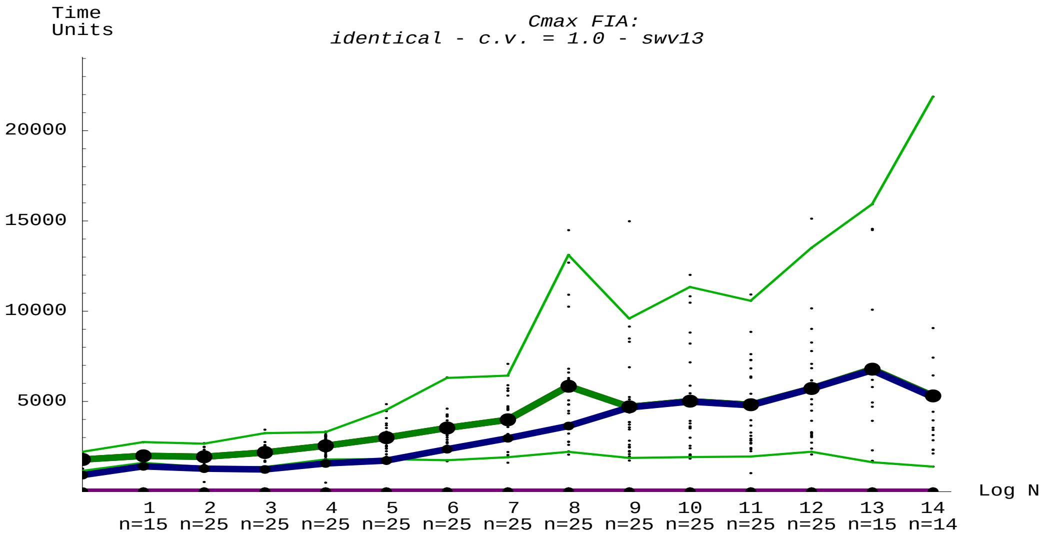
A.1.3 C.V.= 1.0

$C_{\max}$ FIA

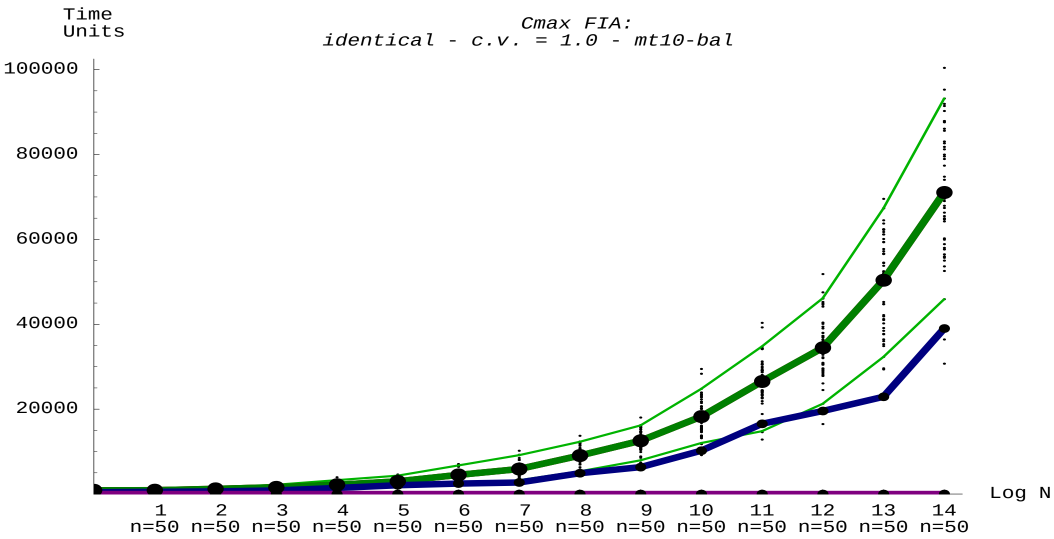




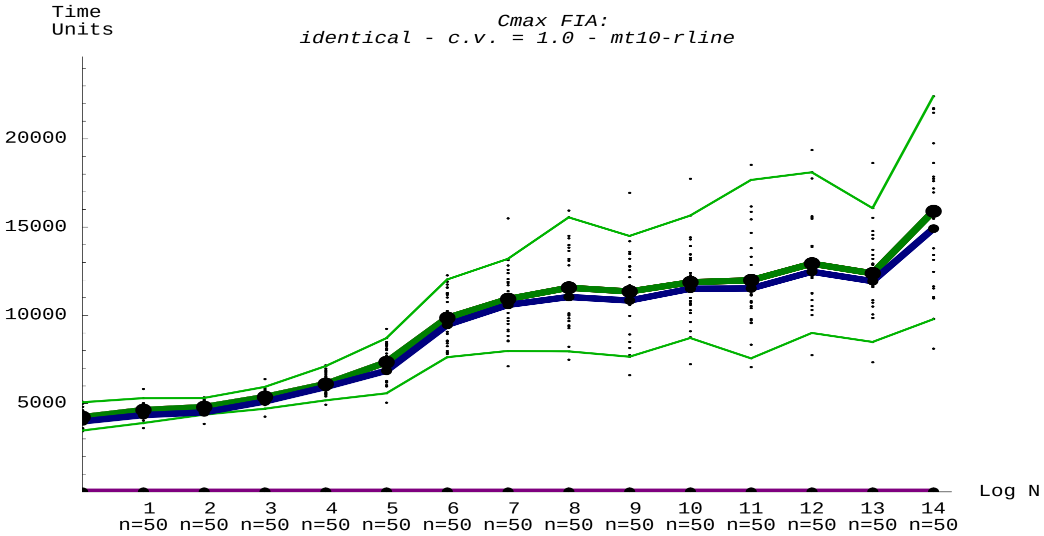
Cmax FIA:
identical - c.v. = 1.0 - abz9

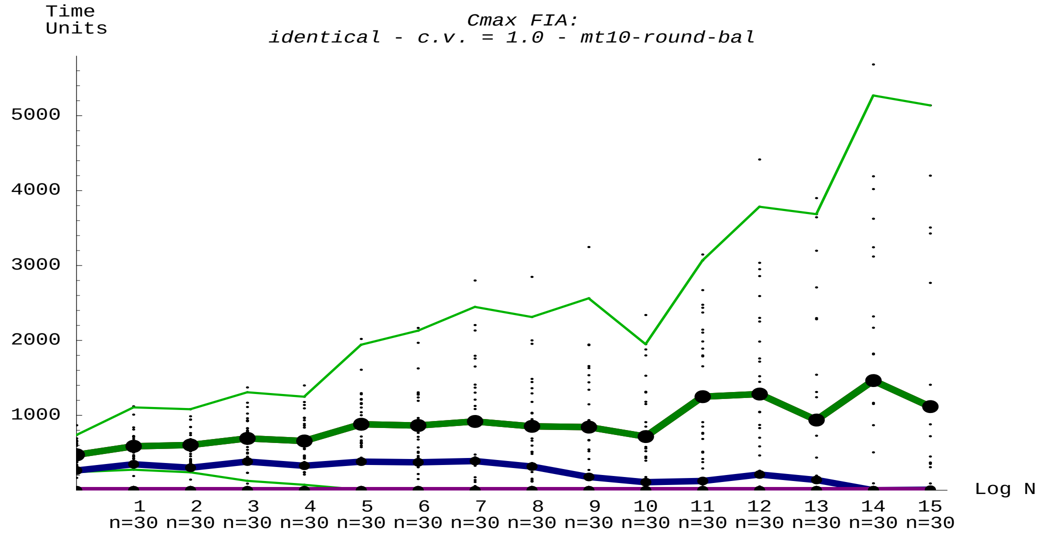


Cmax FIA:
identical - c.v. = 1.0 - swv13

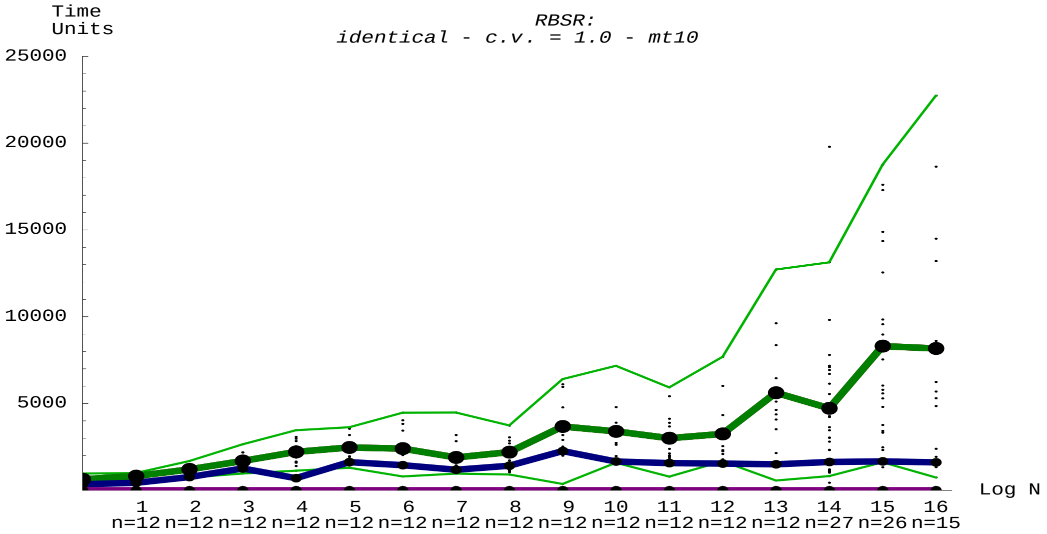


Cmax FIA:
identical - c.v. = 1.0 - mt10-bal

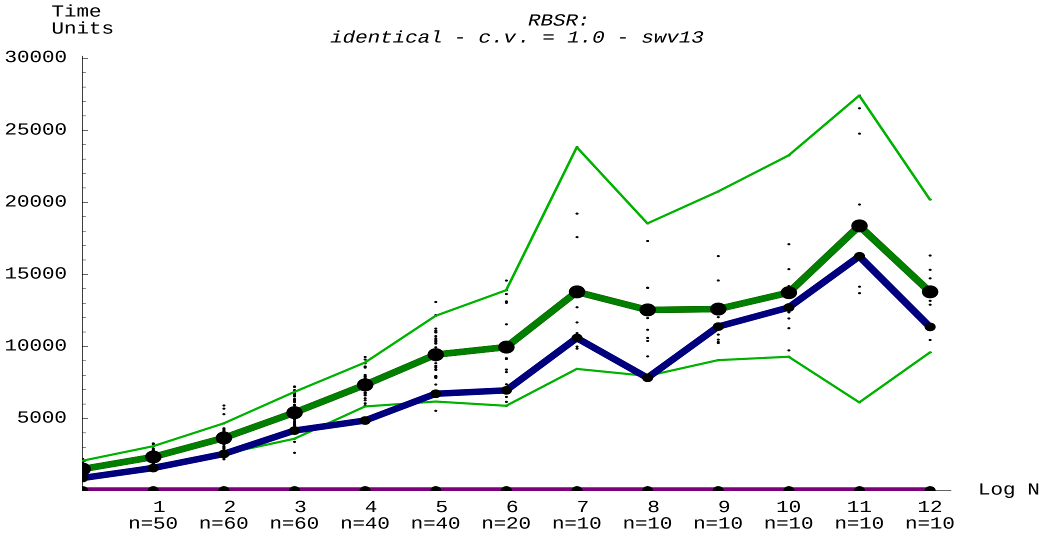

Time
Units
Cmax FIA:
identical - c.v. = 1.0 - mt10-rline
20000
15000
10000
5000
Log N
1
n=50
2
n=50
3
n=50
4
n=50
5
n=50
6
n=50
7
n=50
8
n=50
9
n=50
10
n=50
11
n=50
12
n=50
13
n=50
14
n=50


Time
Units
Cmax FIA:
identical - c.v. = 1.0 - mt10-round-bal
5000
4000
3000
2000
1000
Log N
1
n=30
2
n=30
3
n=30
4
n=30
5
n=30
6
n=30
7
n=30
8
n=30
9
n=30
10
n=30
11
n=30
12
n=30
13
n=30
14
n=30
15
n=30

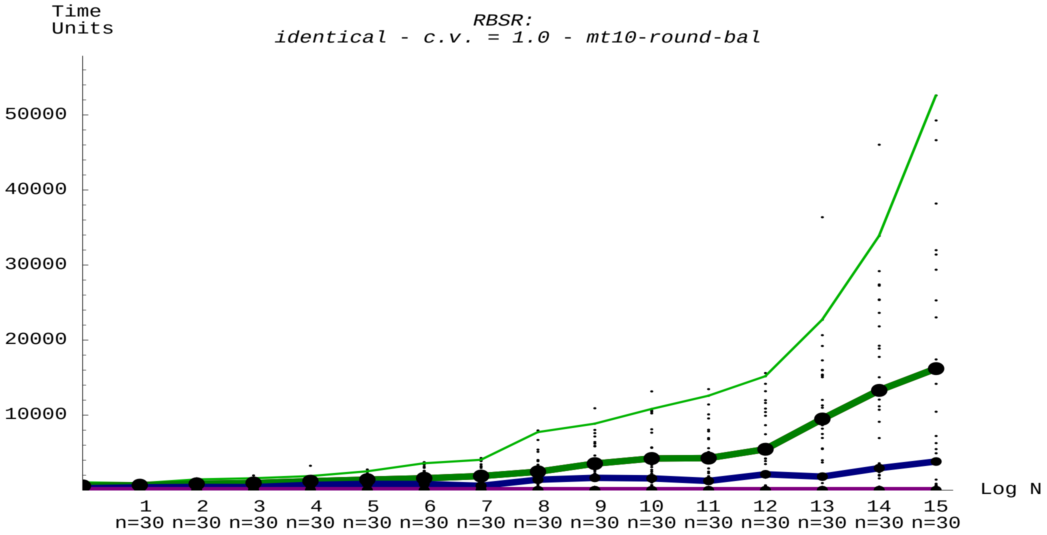
RBSR



RBSR:
identical - c.v. = 1.0 - mt10



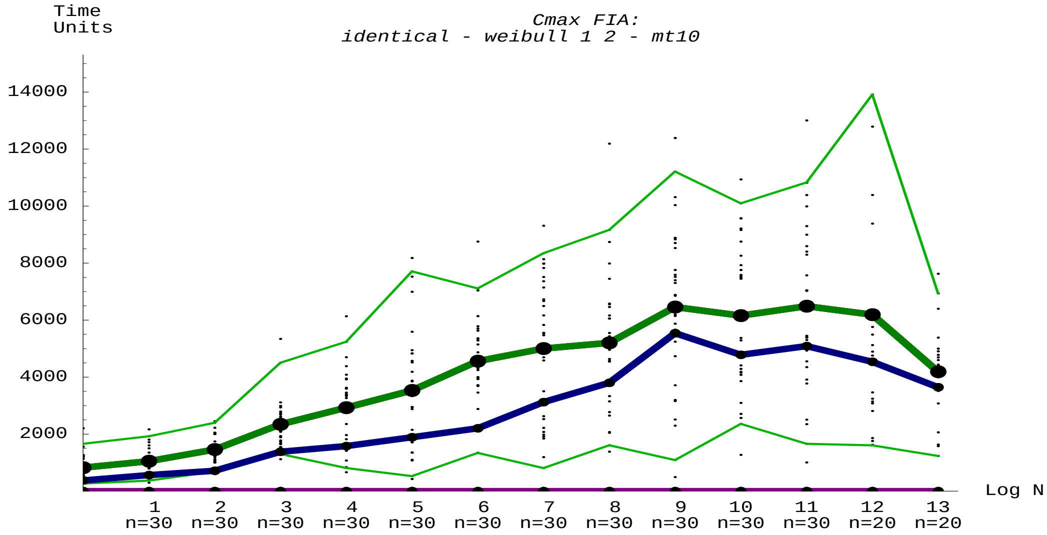
RBSR:
identical - c.v. = 1.0 - swv13



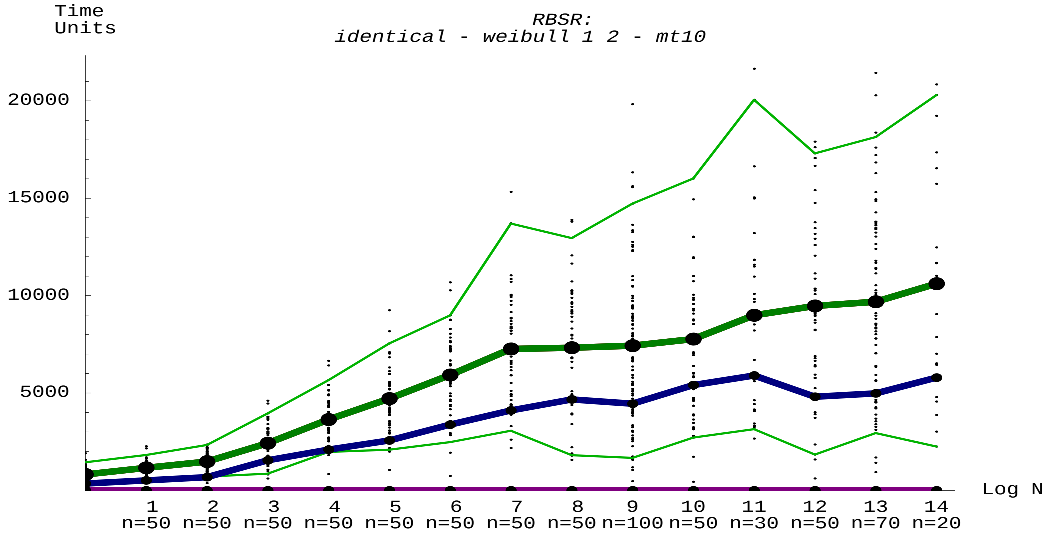
RBSR:
identical - c.v. = 1.0 - mt10-round-bal

A.1.4 Weibull 1/2

$C_{\max}$ FIA

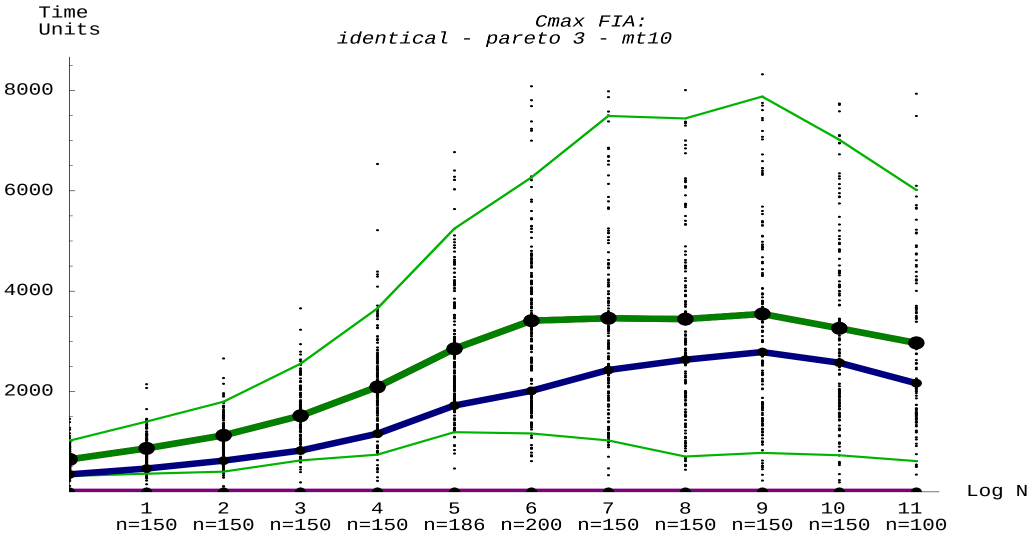


RBSR

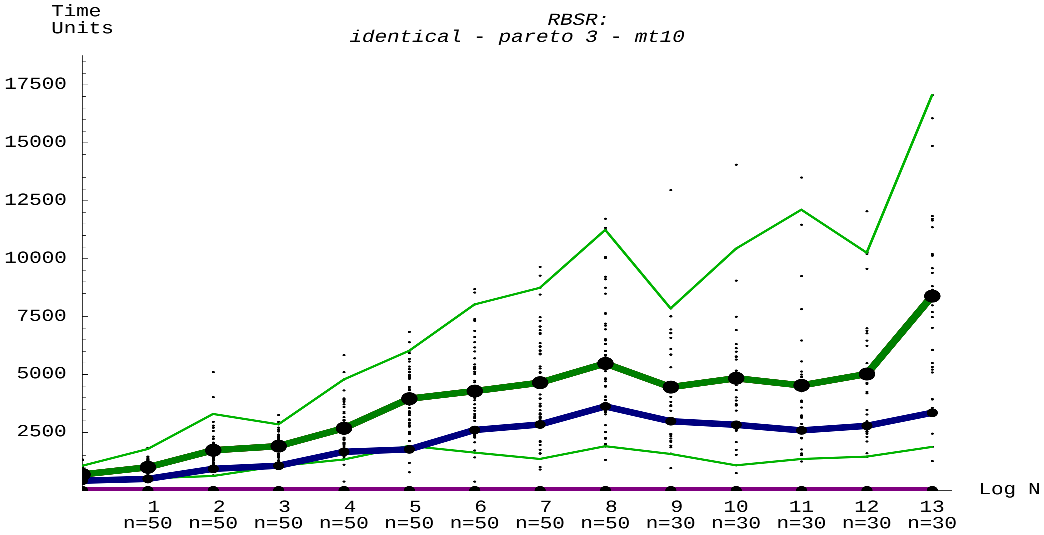


A.1.5 Pareto 3

$C_{\max}$ **FIA**

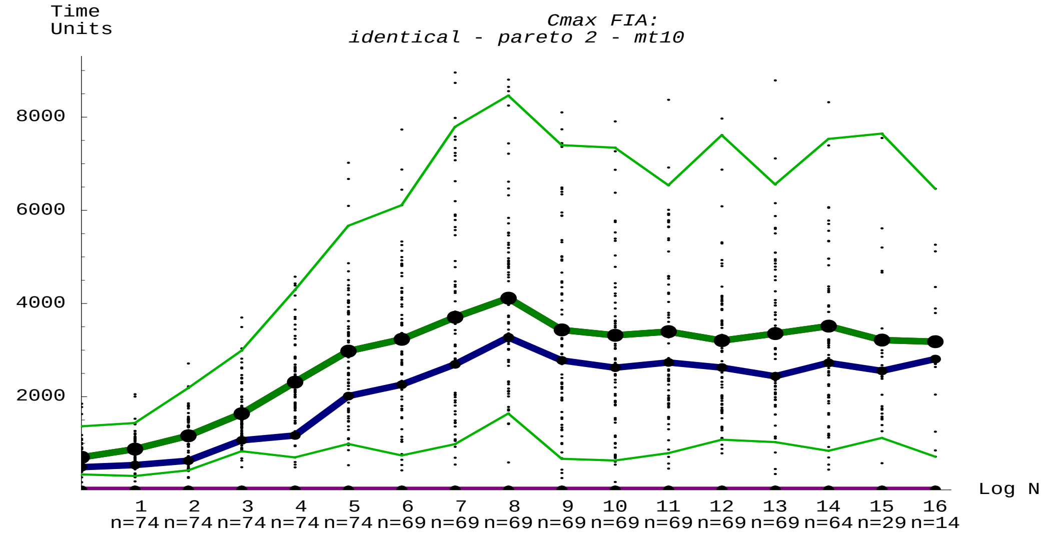


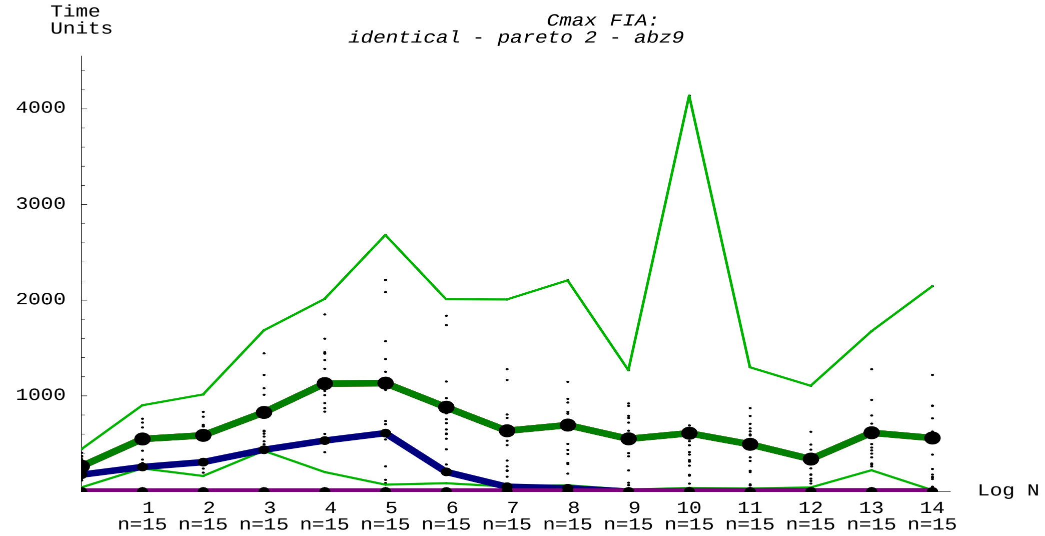
RBSR

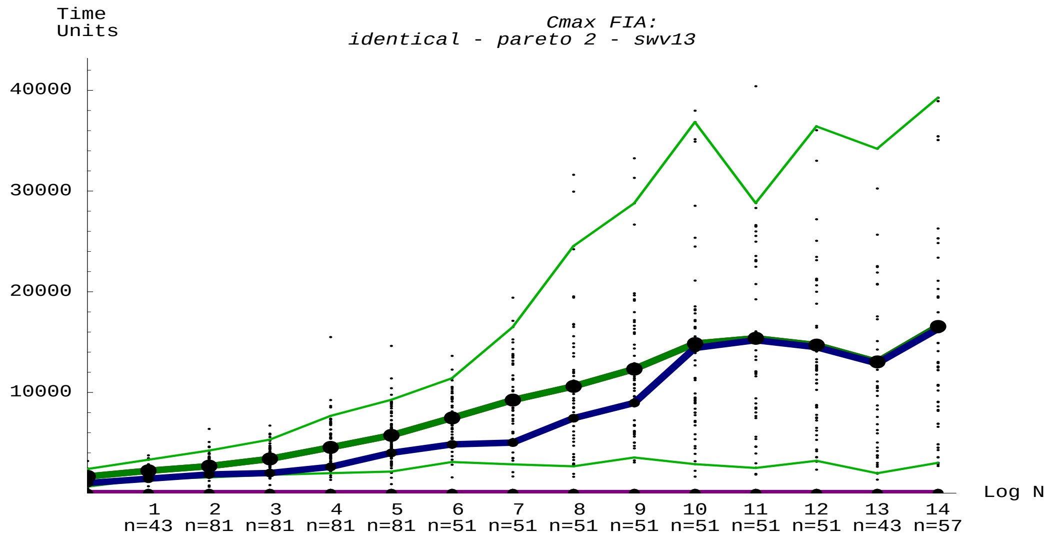


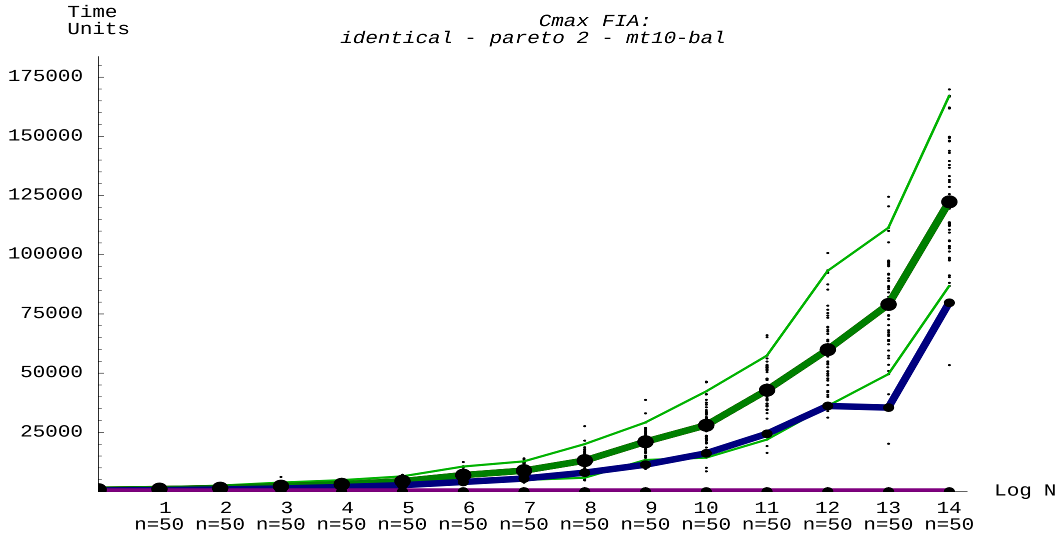
A.1.6 Pareto 2

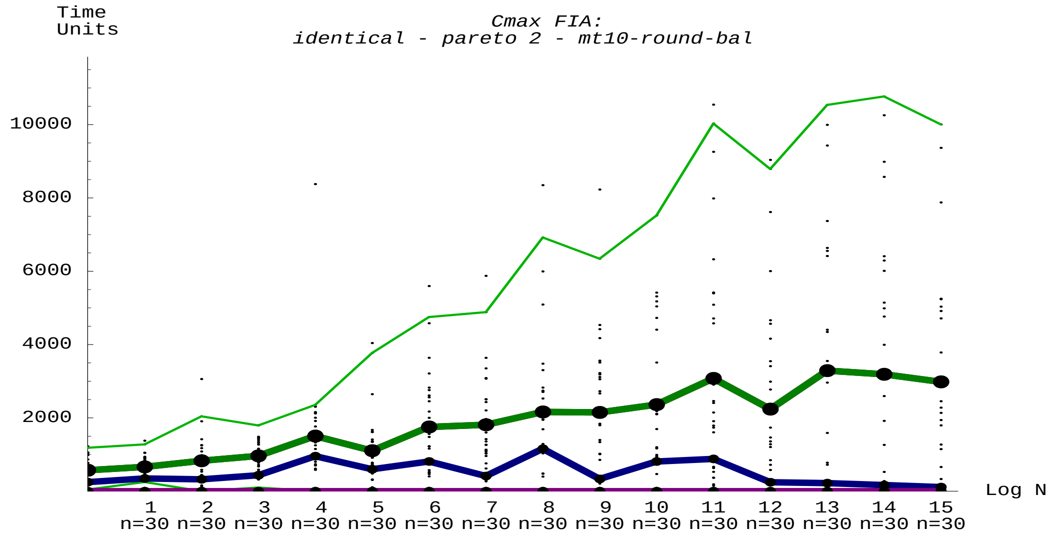
$C_{\max}$ **FIA**



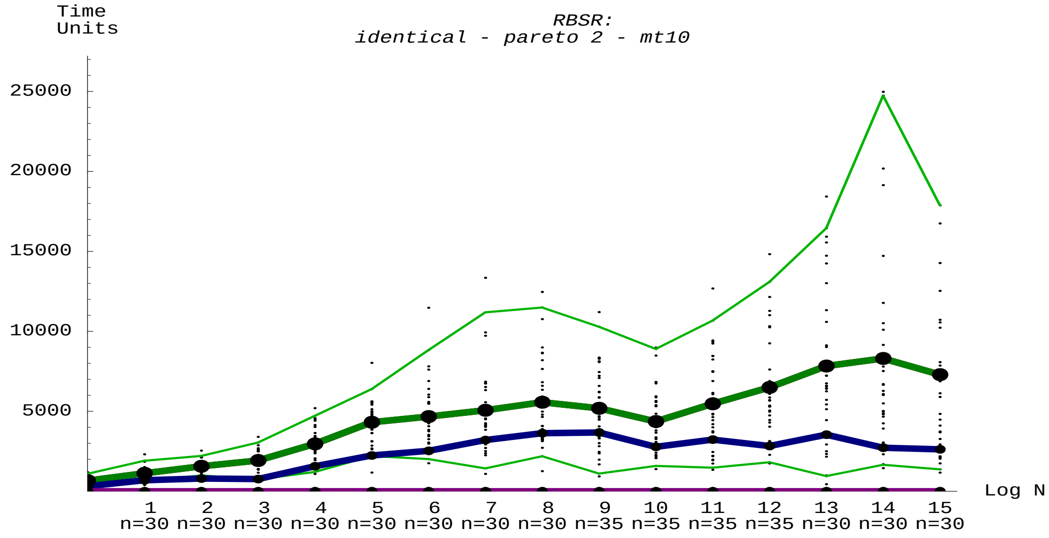


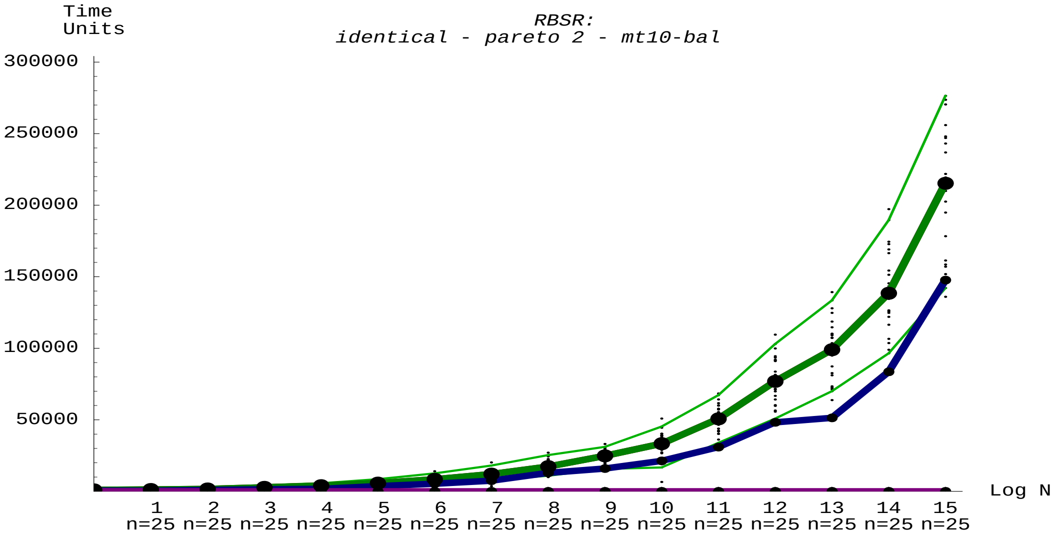

Time
Units
Cmax FIA:
identical - pareto 2 - swv13
40000
30000
20000
10000
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14
n=43 n=81 n=81 n=81 n=81 n=51 n=51 n=51 n=51 n=51 n=51 n=51 n=43 n=57

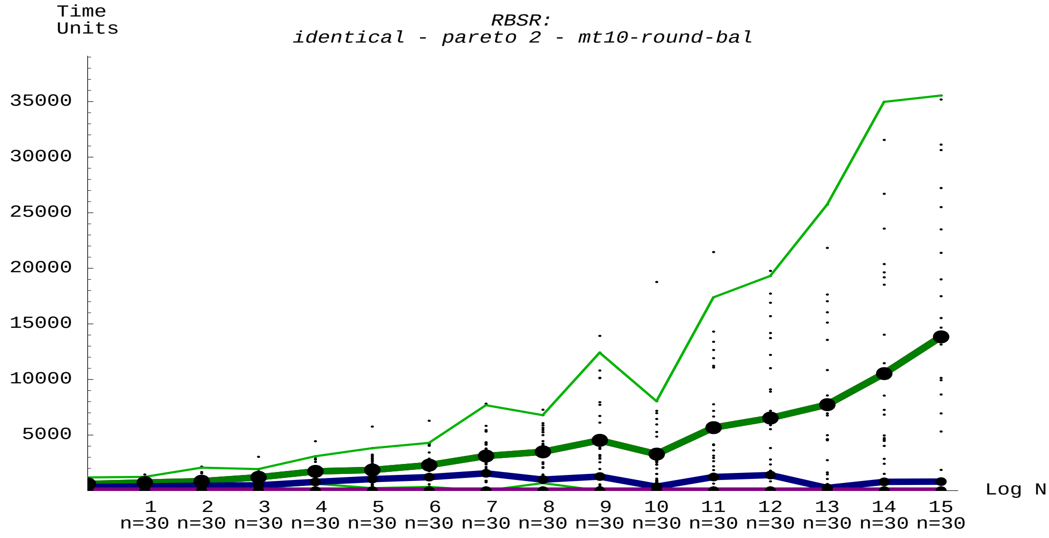

Time
Units
Cmax FIA:
identical - pareto 2 - mt10-bal
175000
150000
125000
100000
75000
50000
25000
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14
n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50


Time
Units
Cmax FIA:
identical - pareto 2 - mt10-round-bal
10000
8000
6000
4000
2000
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30

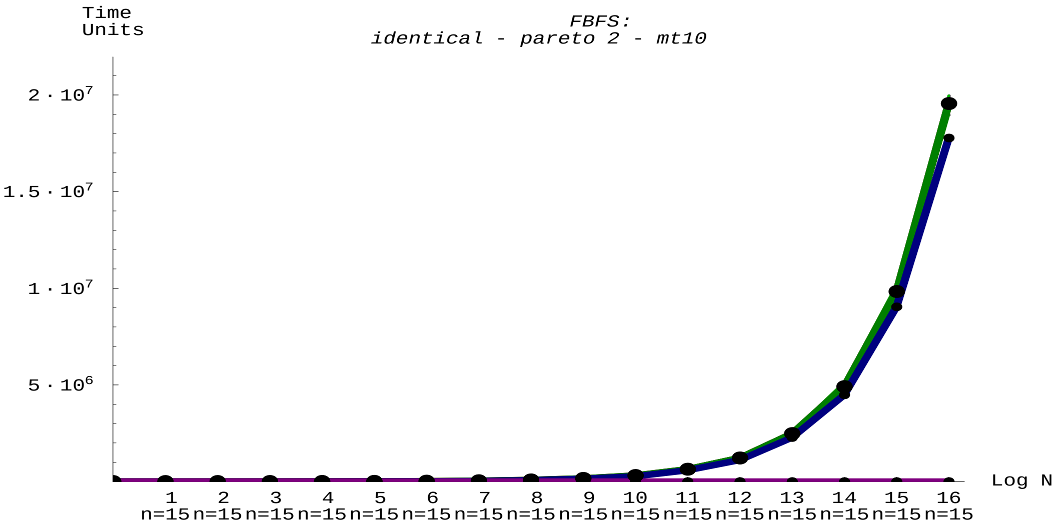
RBSR


Time
Units
RBSR:
identical - pareto 2 - mt10
25000
20000
15000
10000
5000
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=35 n=35 n=35 n=35 n=30 n=30 n=30

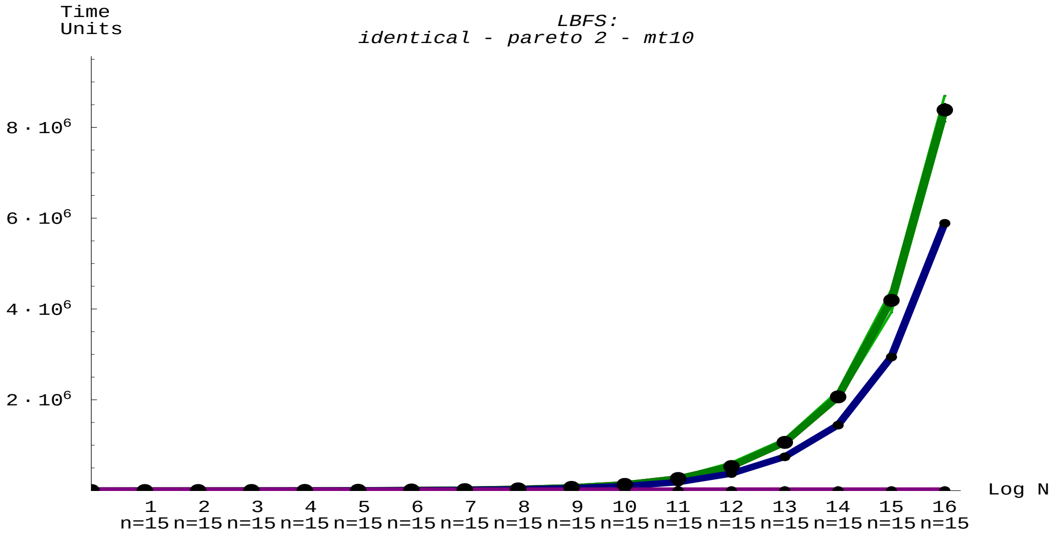

Time
Units
RBSR:
identical - pareto 2 - mt10-bal
300000
250000
200000
150000
100000
50000
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25

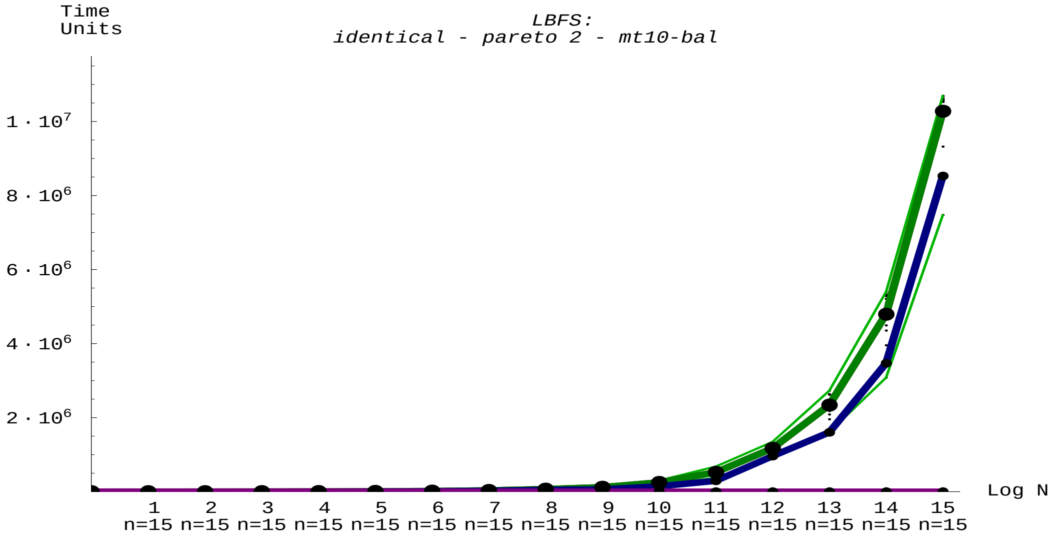

Time
Units
RBSR:
identical - pareto 2 - mt10-round-bal
35000
30000
25000
20000
15000
10000
5000
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30 n=30

FBFS


Time
Units
FBFS:
identical - pareto 2 - mt10
$2 \cdot 10^7$
$1.5 \cdot 10^7$
$1 \cdot 10^7$
$5 \cdot 10^6$
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15

LBFS

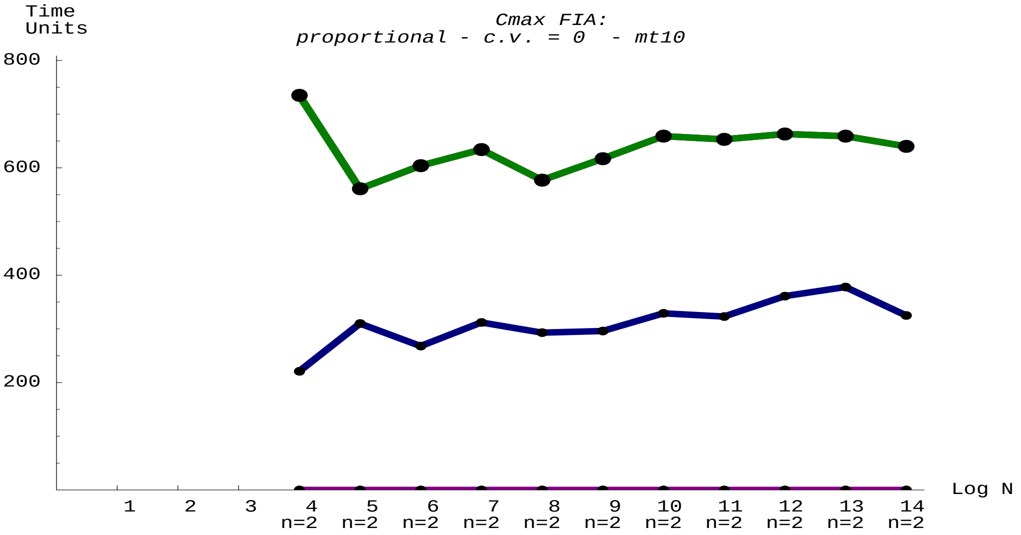

Time
Units
LBFS:
identical - pareto 2 - mt10
$8 \cdot 10^6$
$6 \cdot 10^6$
$4 \cdot 10^6$
$2 \cdot 10^6$
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15

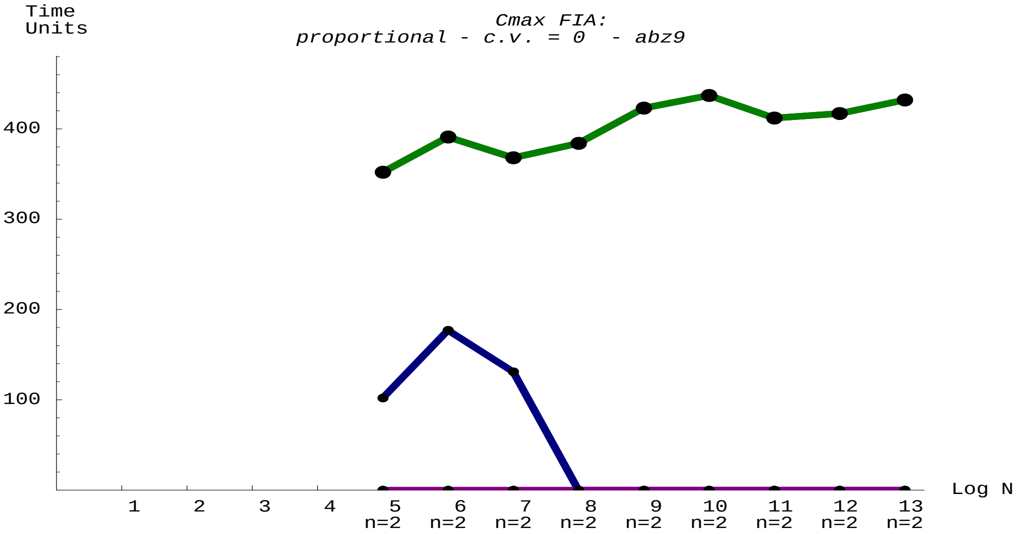

Time
Units
LBFS:
identical - pareto 2 - mt10-bal
1 · 10^7
8 · 10^6
6 · 10^6
4 · 10^6
2 · 10^6
Log N
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15 n=15

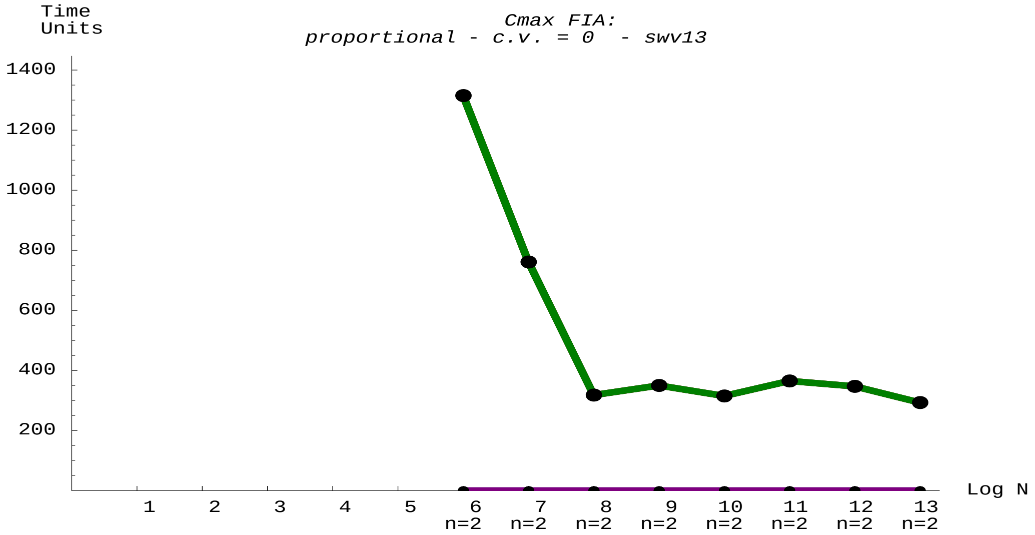
A.2 The $R \ll J$ Proportional Case

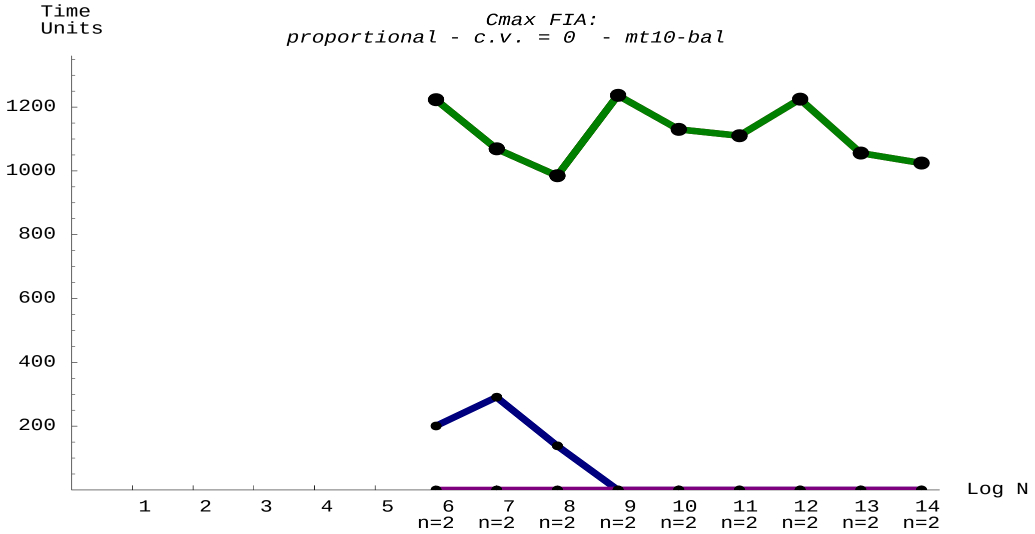
A.2.1 C.V. = 0

$C_{\max}$ **FIA**

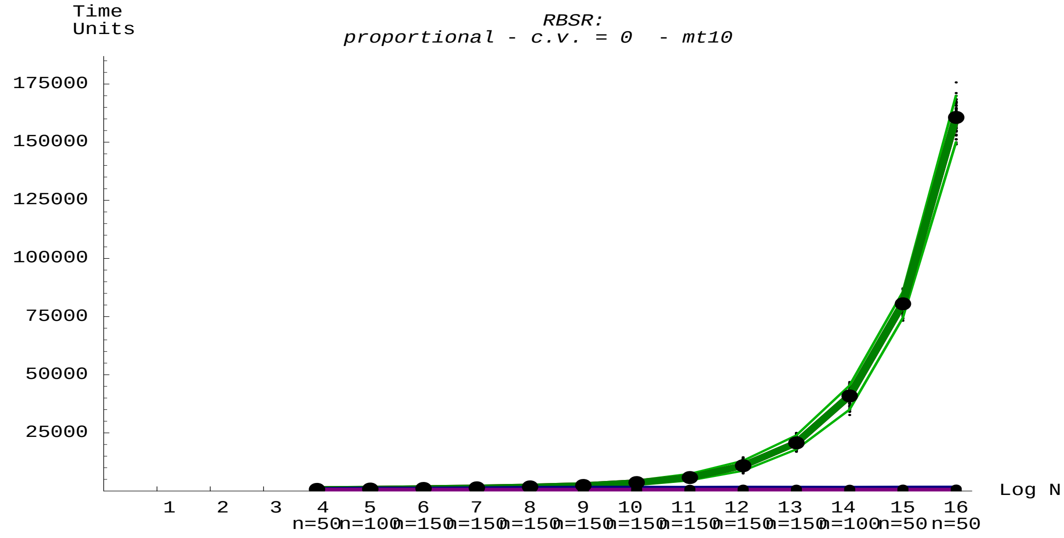


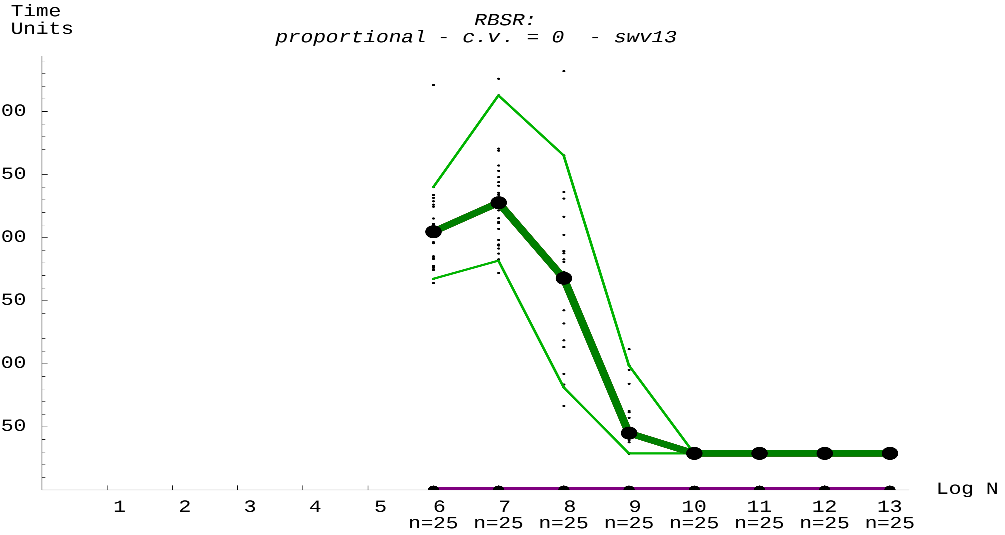


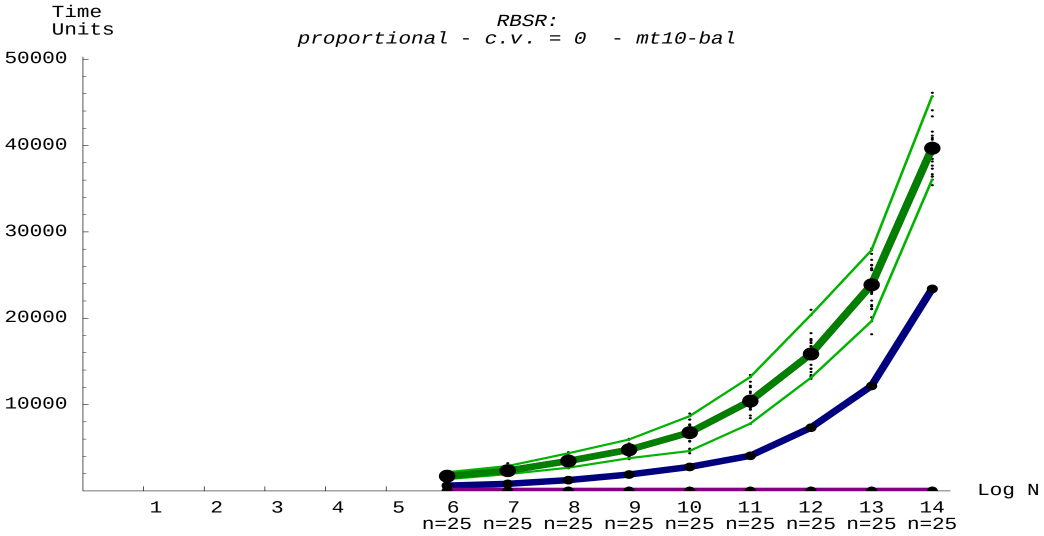

Time
Units
Cmax FIA:
proportional - c.v. = 0 - swv13
1400
1200
1000
800
600
400
200
Log N
1
2
3
4
5
6
n=2
7
n=2
8
n=2
9
n=2
10
n=2
11
n=2
12
n=2
13
n=2


Time
Units
Cmax FIA:
proportional - c.v. = 0 - mt10-bal
1200
1000
800
600
400
200
Log N
1
2
3
4
5
6
n=2
7
n=2
8
n=2
9
n=2
10
n=2
11
n=2
12
n=2
13
n=2
14
n=2

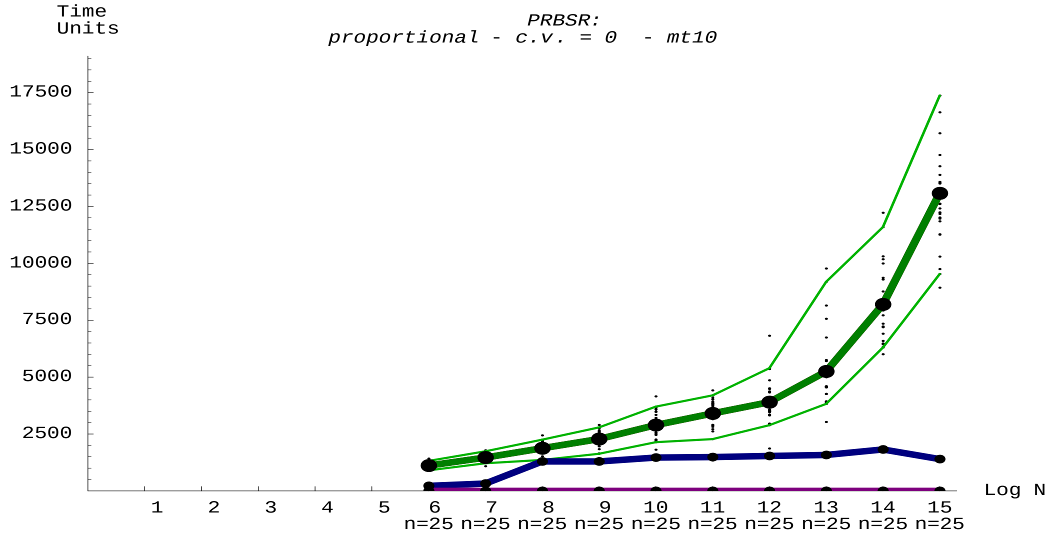
RBSR

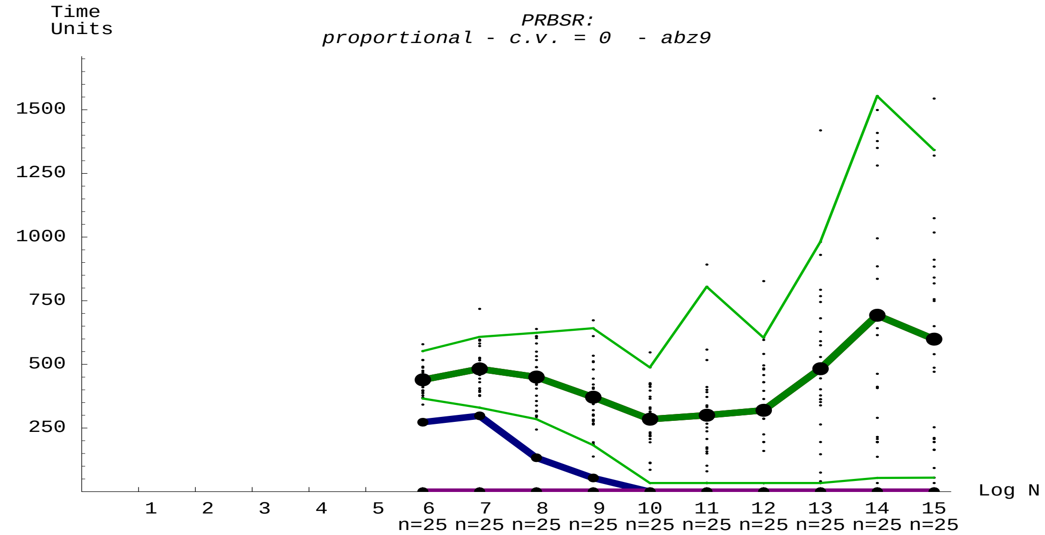

Time
Units
RBSR:
proportional - c.v. = 0 - mt10
175000
150000
125000
100000
75000
50000
25000
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Log N
n=50
n=100
n=150
n=150
n=150
n=150
n=150
n=150
n=150
n=150
n=100
n=50
n=50


Time
Units
RBSR:
proportional - c.v. = 0 - swv13
1500
1250
1000
750
500
250
1
2
3
4
5
6
7
8
9
10
11
12
13
Log N
n=25
n=25
n=25
n=25
n=25
n=25
n=25
n=25

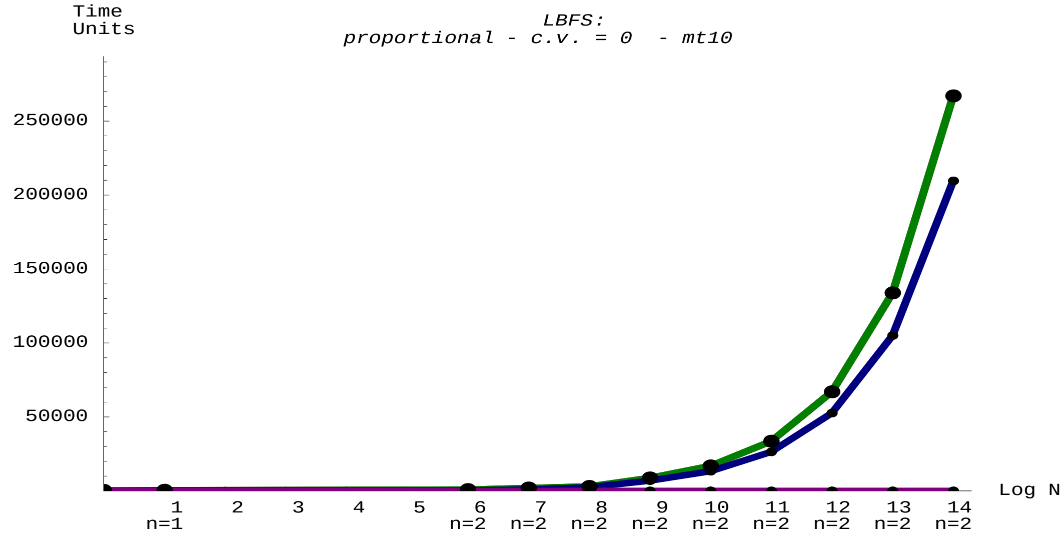

Time
Units
RBSR:
proportional - c.v. = 0 - mt10-bal
50000
40000
30000
20000
10000
1
2
3
4
5
6
7
8
9
10
11
12
13
14
Log N
n=25
n=25
n=25
n=25
n=25
n=25
n=25
n=25
n=25

PRBSR


Time
Units
PRBSR:
proportional - c.v. = 0 - mt10
17500
15000
12500
10000
7500
5000
2500
Log N
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25

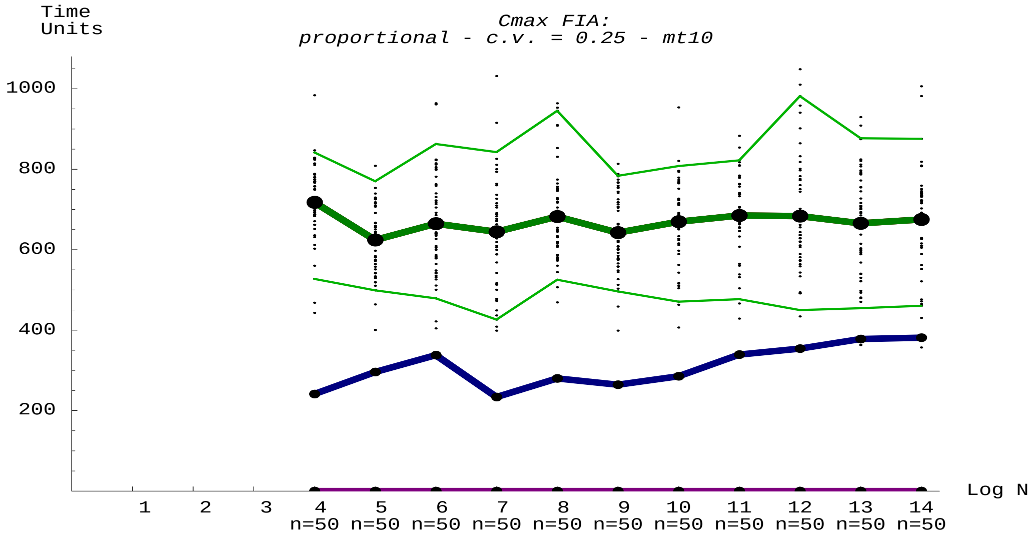

Time
Units
PRBSR:
proportional - c.v. = 0 - abz9
1500
1250
1000
750
500
250
Log N
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25 n=25

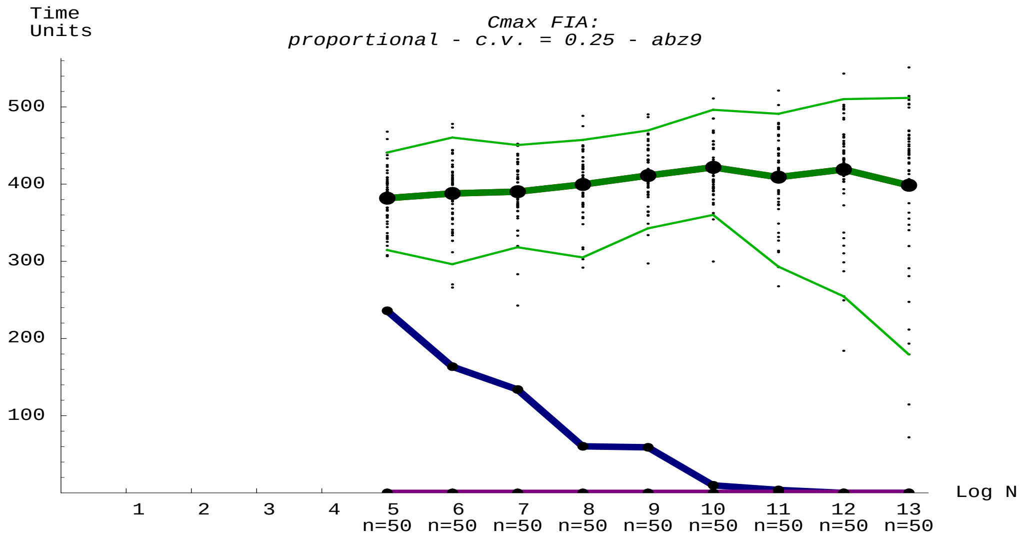
LBFS

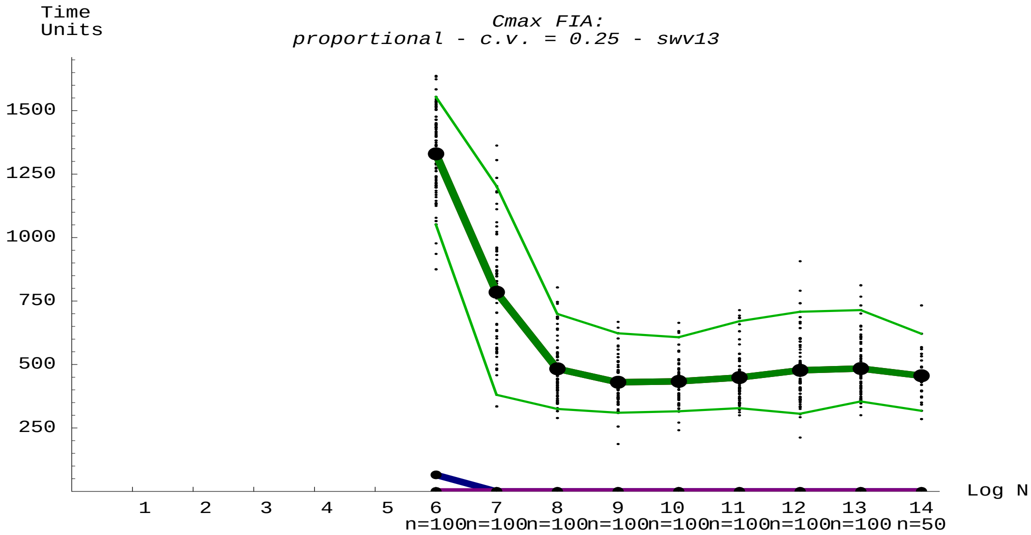


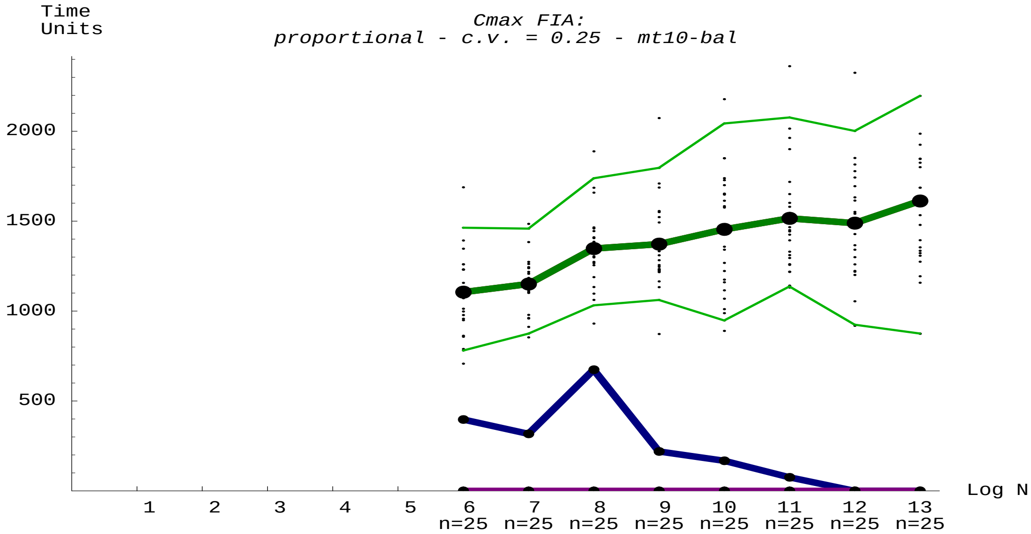
A.2.2 C.V.= 0.25

$C_{\max}$ FIA

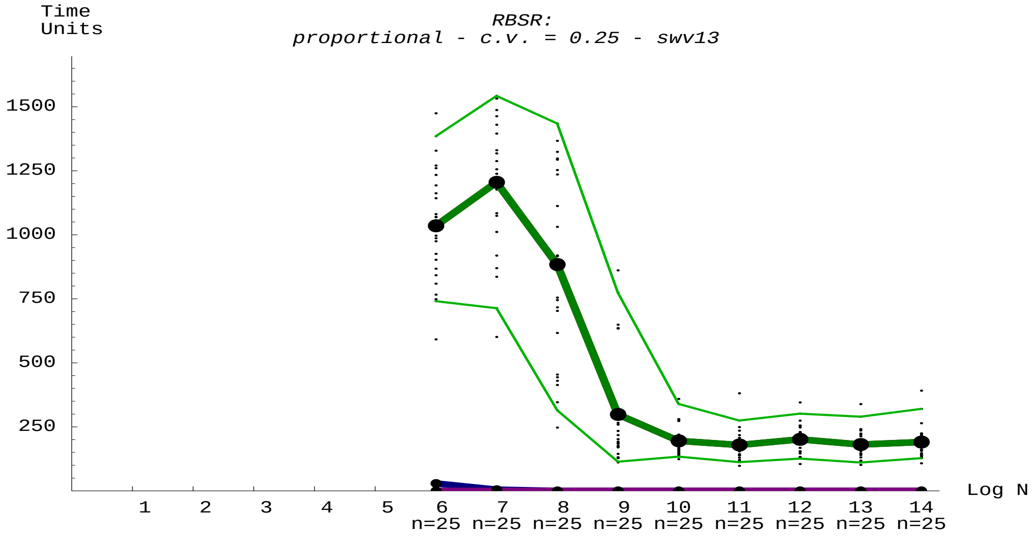



Time
Units
Cmax FIA:
proportional - c.v. = 0.25 - abz9
500
400
300
200
100
Log N
1
2
3
4
5
n=50
6
n=50
7
n=50
8
n=50
9
n=50
10
n=50
11
n=50
12
n=50
13
n=50


Time
Units
Cmax FIA:
proportional - c.v. = 0.25 - swv13
1500
1250
1000
750
500
250
Log N
1
2
3
4
5
6
n=100
7
n=100
8
n=100
9
n=100
10
n=100
11
n=100
12
n=100
13
n=100
14
n=50

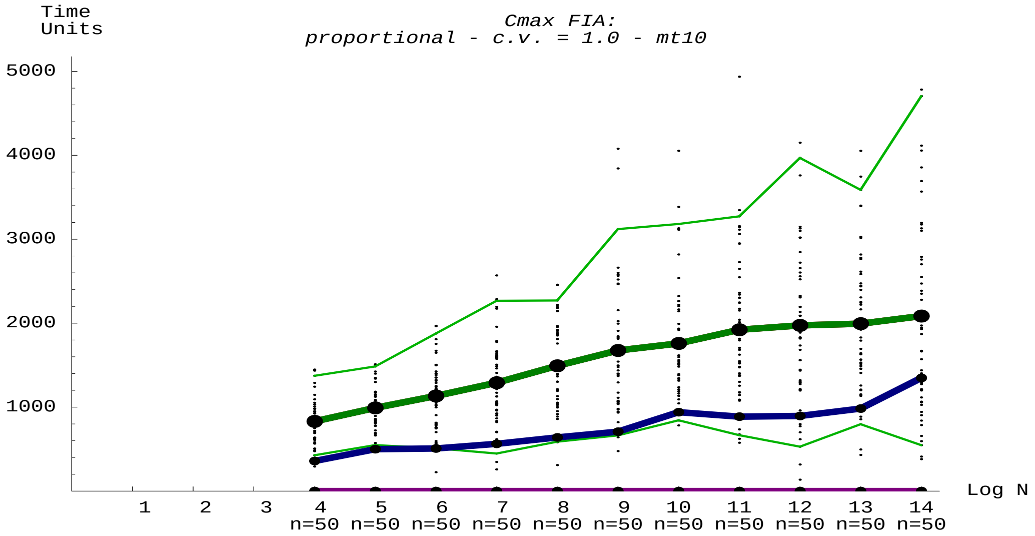

Time
Units
Cmax FIA:
proportional - c.v. = 0.25 - mt10-bal
2000
1500
1000
500
Log N
1
2
3
4
5
6
n=25
7
n=25
8
n=25
9
n=25
10
n=25
11
n=25
12
n=25
13
n=25

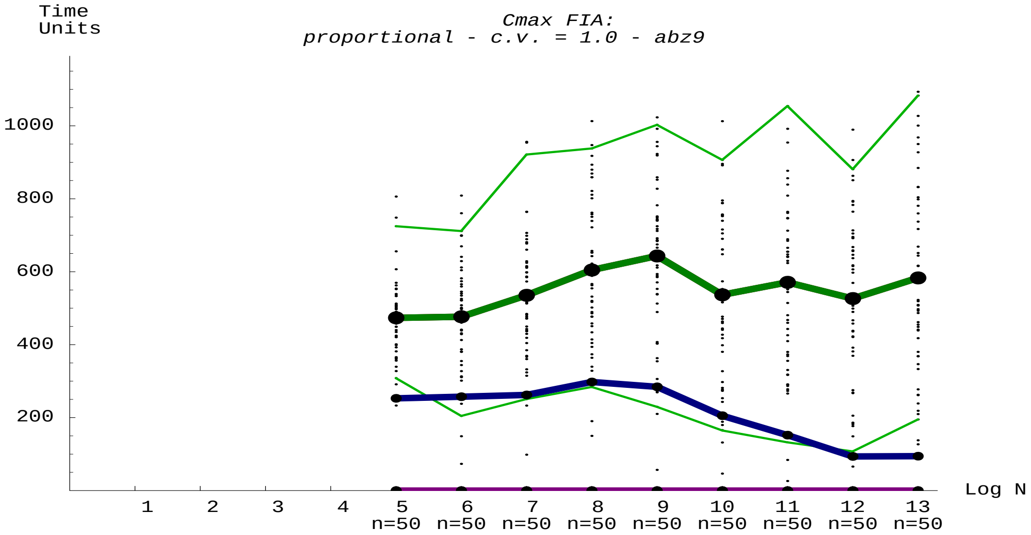
RBSR

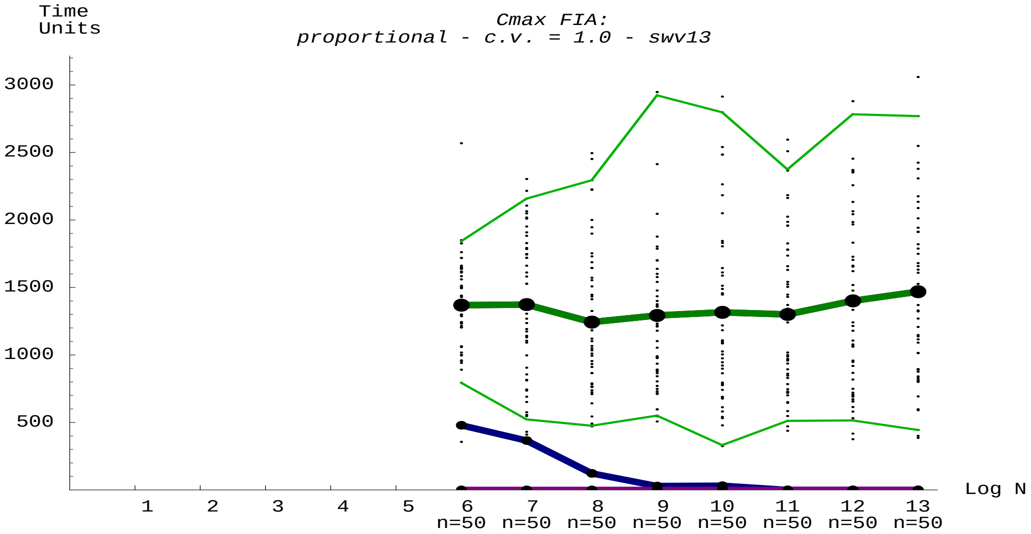


A.2.3 C.V.= 1.0

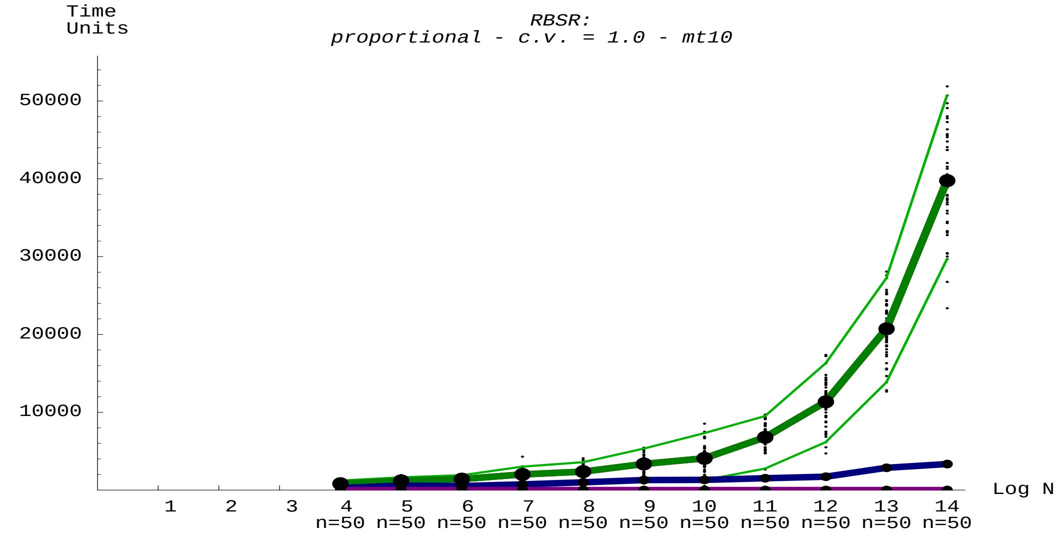
$C_{\max}$ **FIA**

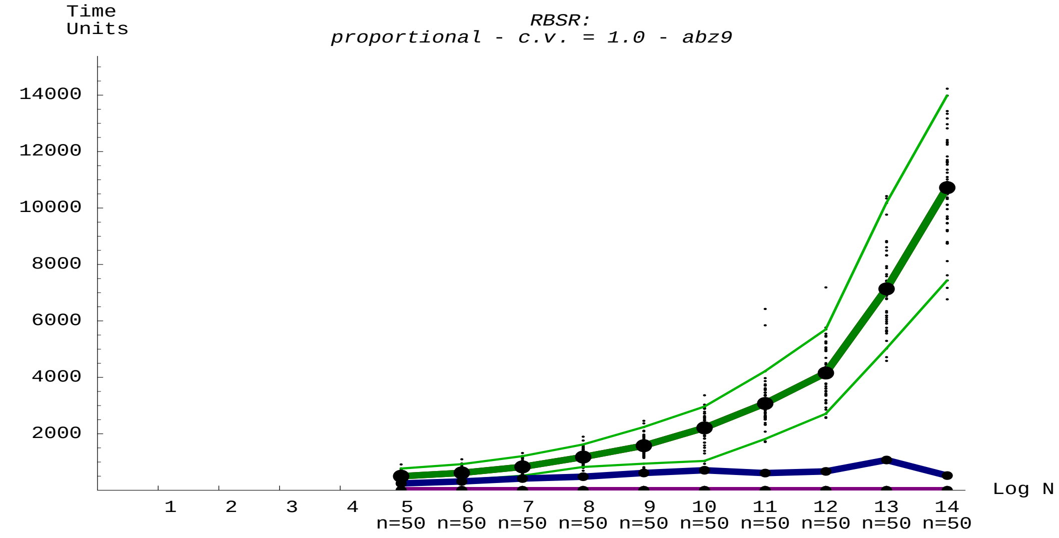


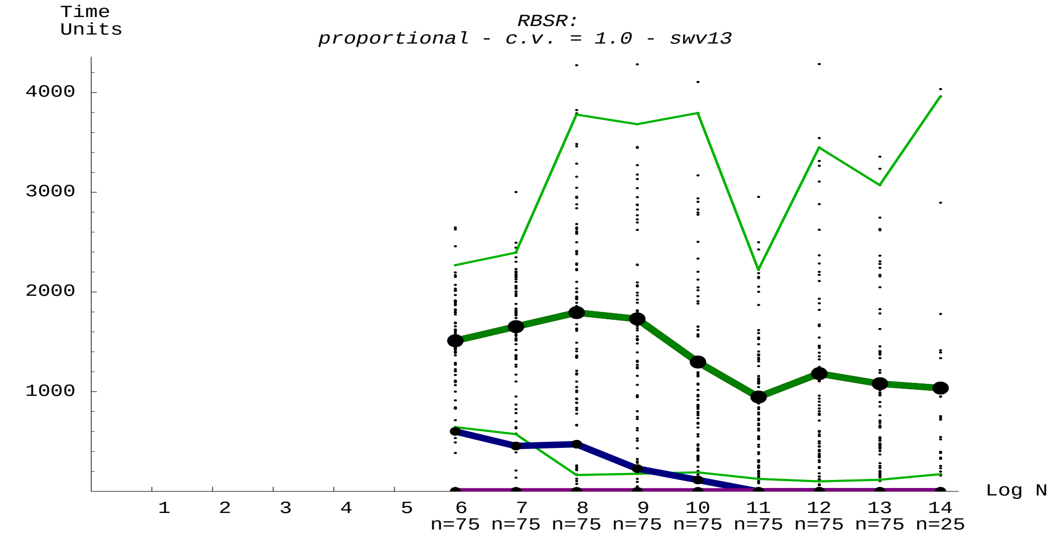

Time Units
Cmax FIA:
proportional - c.v. = 1.0 - abz9
1000
800
600
400
200
Log N
1
2
3
4
5 n=50
6 n=50
7 n=50
8 n=50
9 n=50
10 n=50
11 n=50
12 n=50
13 n=50


Time Units
Cmax FIA:
proportional - c.v. = 1.0 - swv13
3000
2500
2000
1500
1000
500
Log N
1
2
3
4
5
6 n=50
7 n=50
8 n=50
9 n=50
10 n=50
11 n=50
12 n=50
13 n=50

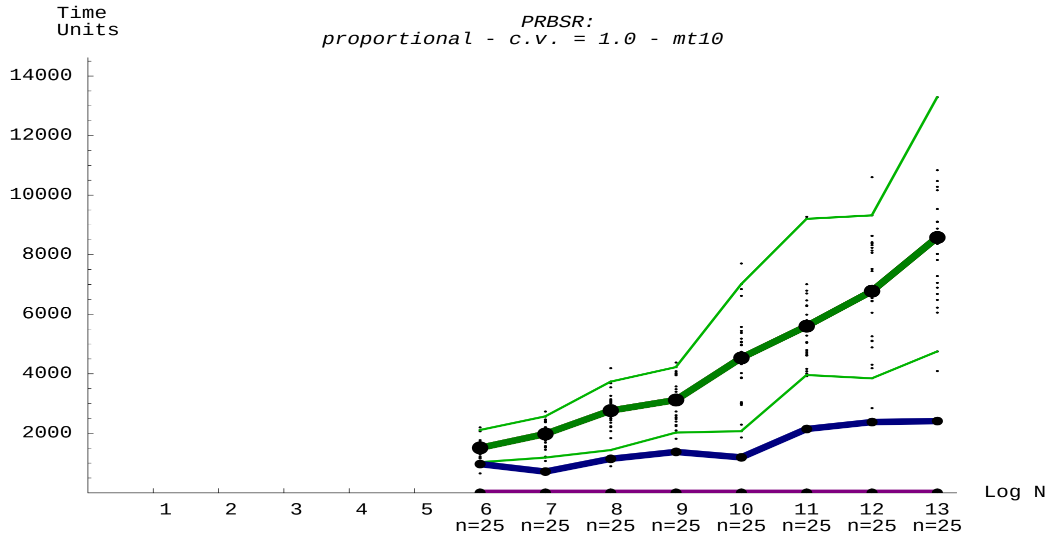
RBSR


Time
Units
RBSR:
proportional - c.v. = 1.0 - mt10
50000
40000
30000
20000
10000
Log N
1
2
3
4
5
6
7
8
9
10
11
12
13
14
n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50

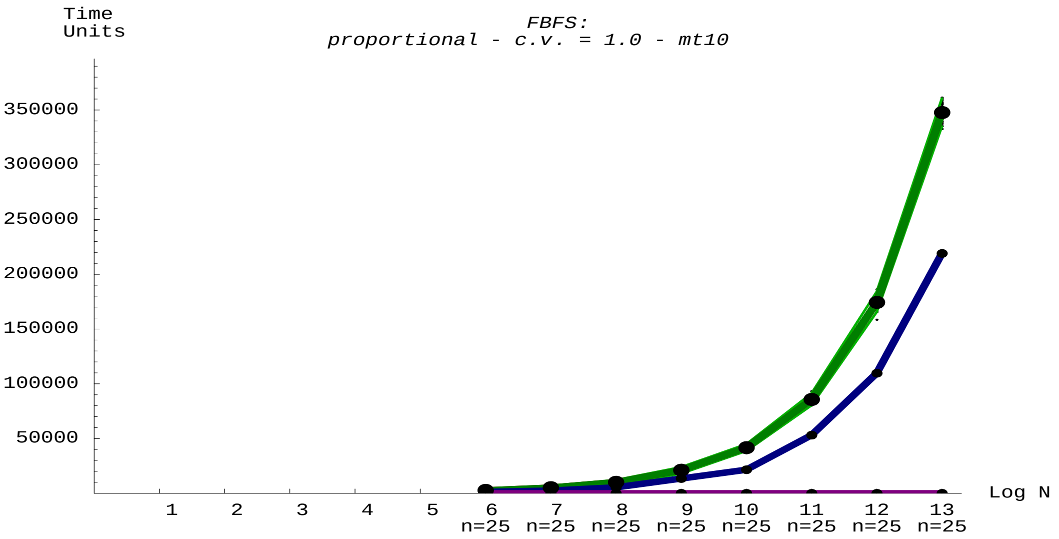

Time
Units
RBSR:
proportional - c.v. = 1.0 - abz9
14000
12000
10000
8000
6000
4000
2000
Log N
1
2
3
4
5
6
7
8
9
10
11
12
13
14
n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50


Time
Units
RBSR:
proportional - c.v. = 1.0 - swv13
4000
3000
2000
1000
Log N
1
2
3
4
5
6
7
8
9
10
11
12
13
14
n=75 n=75 n=75 n=75 n=75 n=75 n=75 n=75 n=25

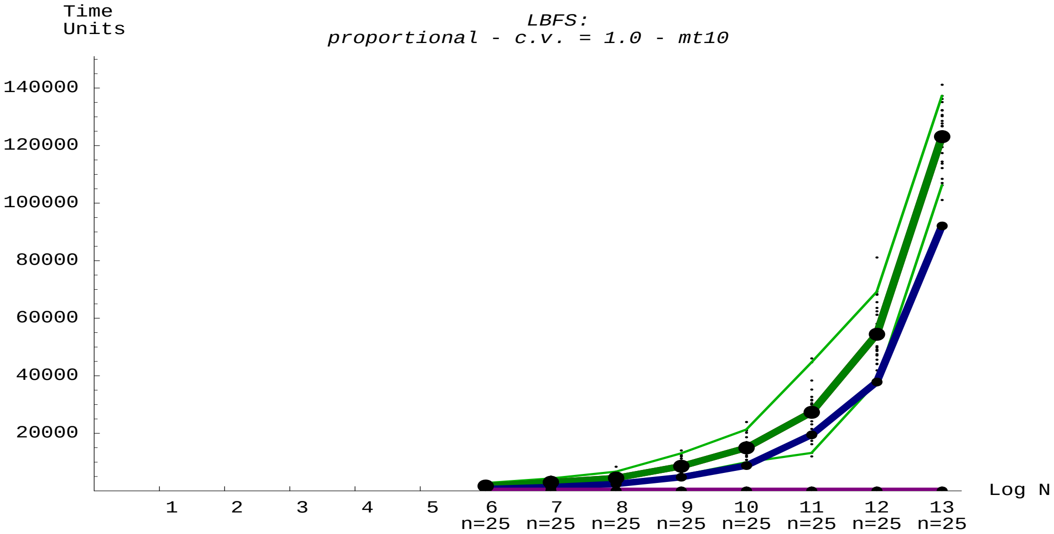
PRBSR


Time
Units
PRBSR:
proportional - c.v. = 1.0 - mt10
14000
12000
10000
8000
6000
4000
2000
Log N
1
2
3
4
5
6
n=25
7
n=25
8
n=25
9
n=25
10
n=25
11
n=25
12
n=25
13
n=25

FBFS

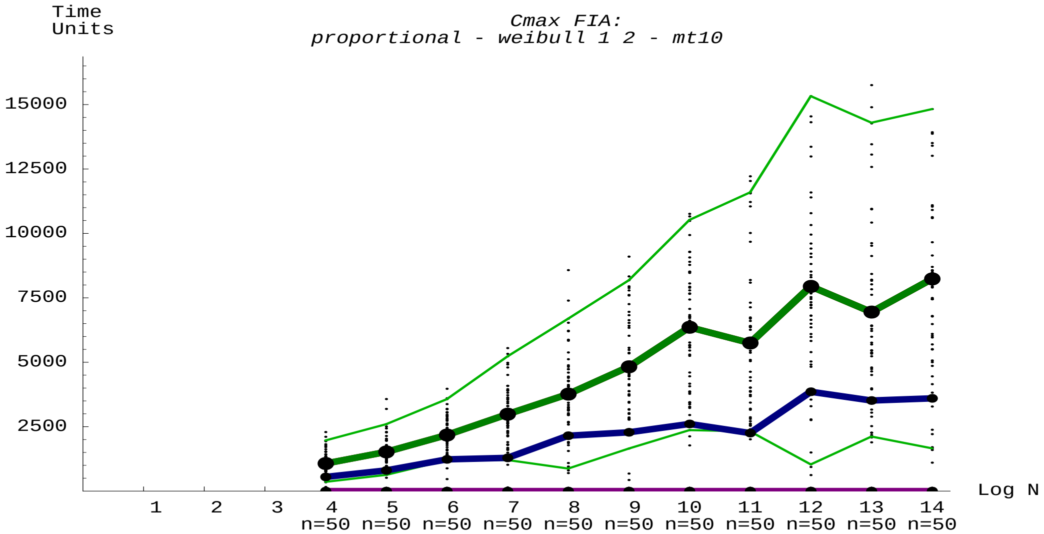

Time
Units
FBFS:
proportional - c.v. = 1.0 - mt10
350000
300000
250000
200000
150000
100000
50000
Log N
1
2
3
4
5
6
n=25
7
n=25
8
n=25
9
n=25
10
n=25
11
n=25
12
n=25
13
n=25

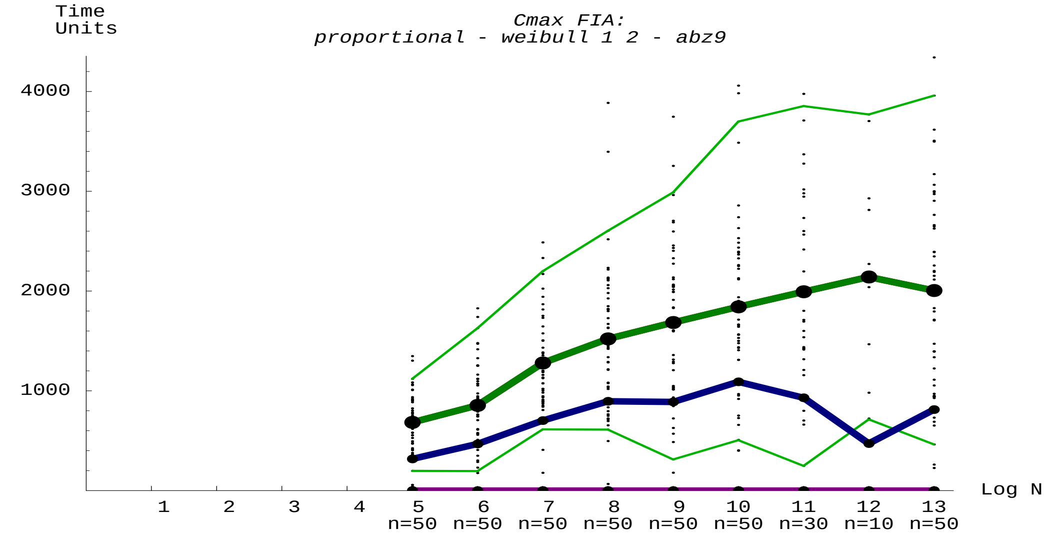
LBFS

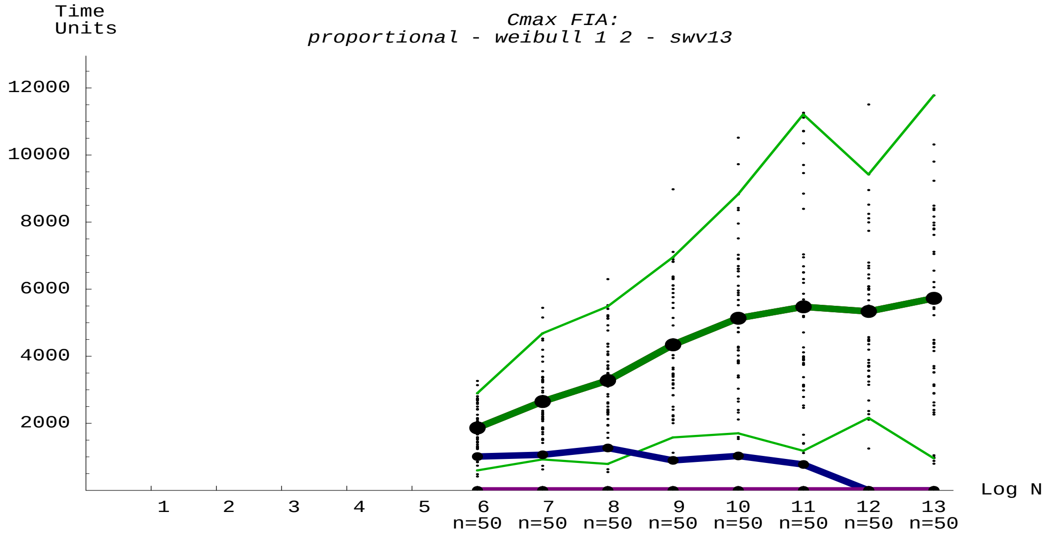


A.2.4 Weibull 1/2

$C_{\max}$ FIA

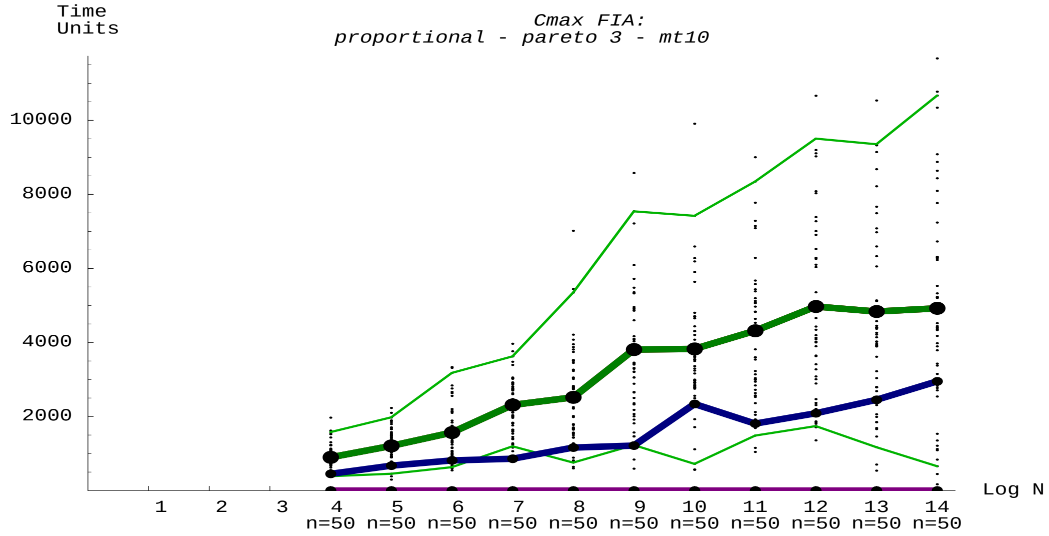


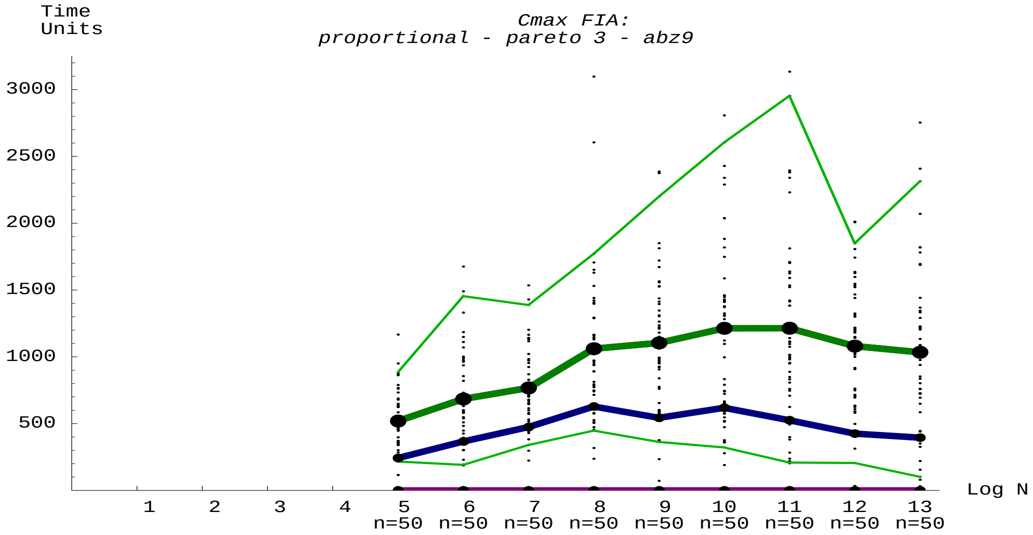

Time
Units
Cmax FIA:
proportional - weibull 1 2 - abz9
4000
3000
2000
1000
Log N
1
2
3
4
5
6
7
8
9
10
11
12
13
n=50
n=50
n=50
n=50
n=50
n=50
n=30
n=10
n=50

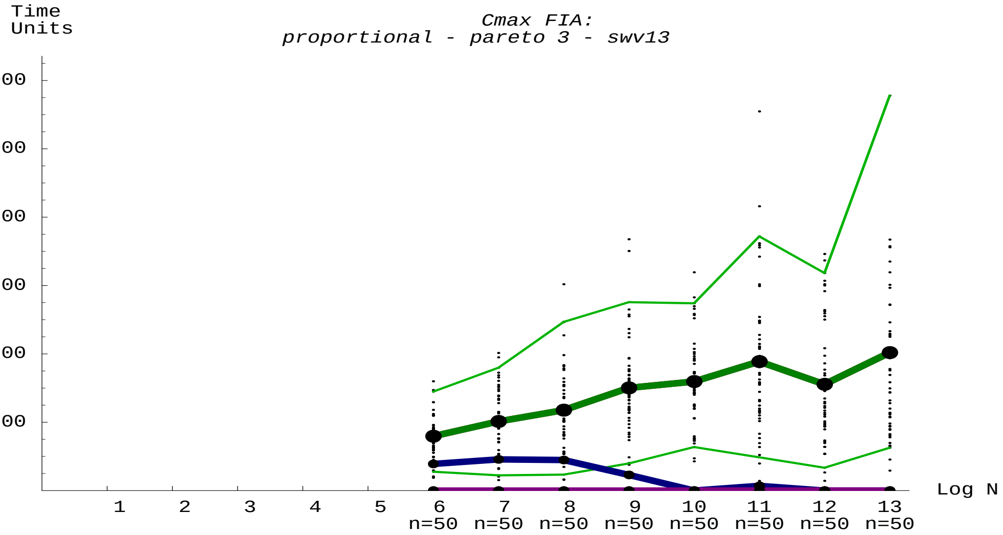

Time
Units
Cmax FIA:
proportional - weibull 1 2 - swv13
12000
10000
8000
6000
4000
2000
Log N
1
2
3
4
5
6
7
8
9
10
11
12
13
n=50
n=50
n=50
n=50
n=50
n=50
n=50
n=50

A.2.5 Pareto 3

$C_{\max}$ **FIA**

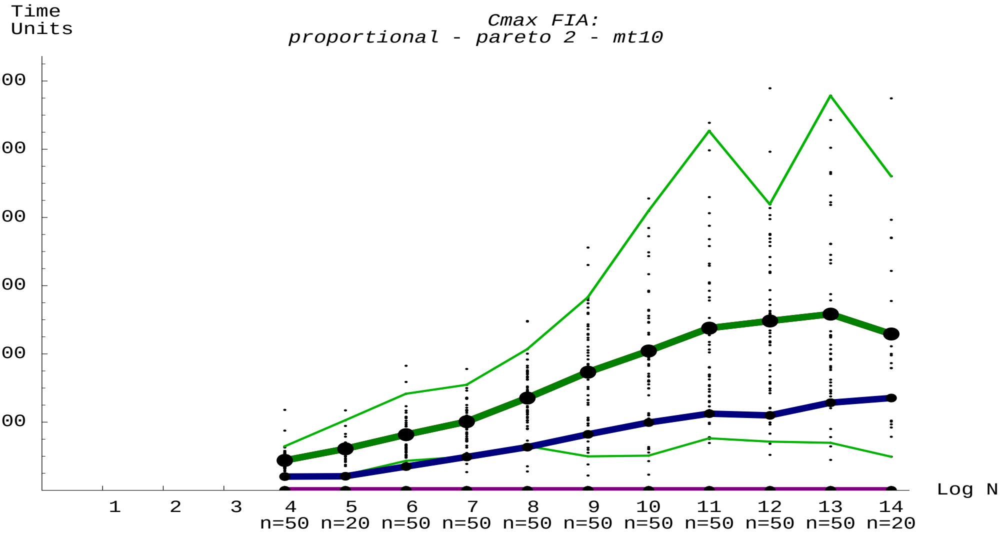


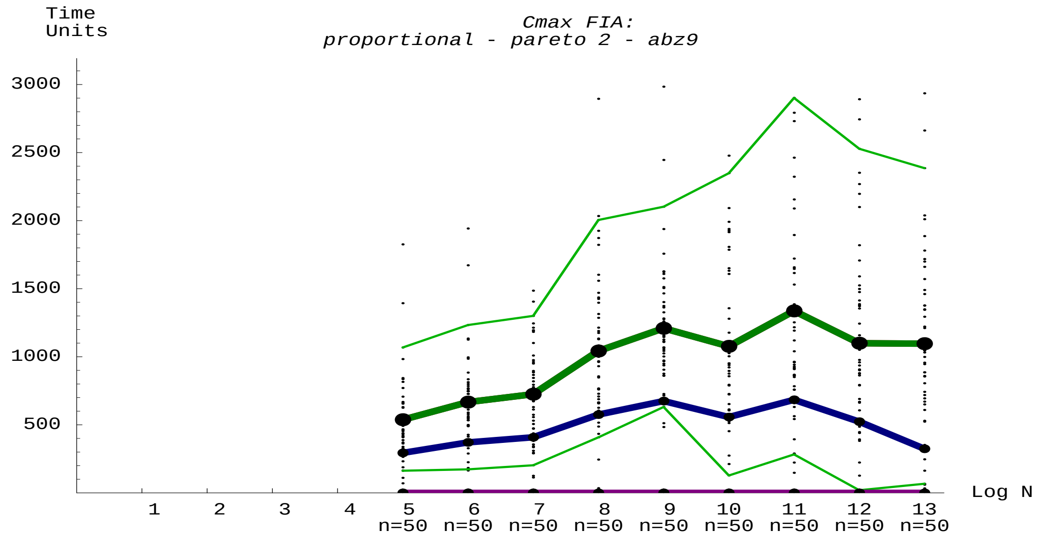


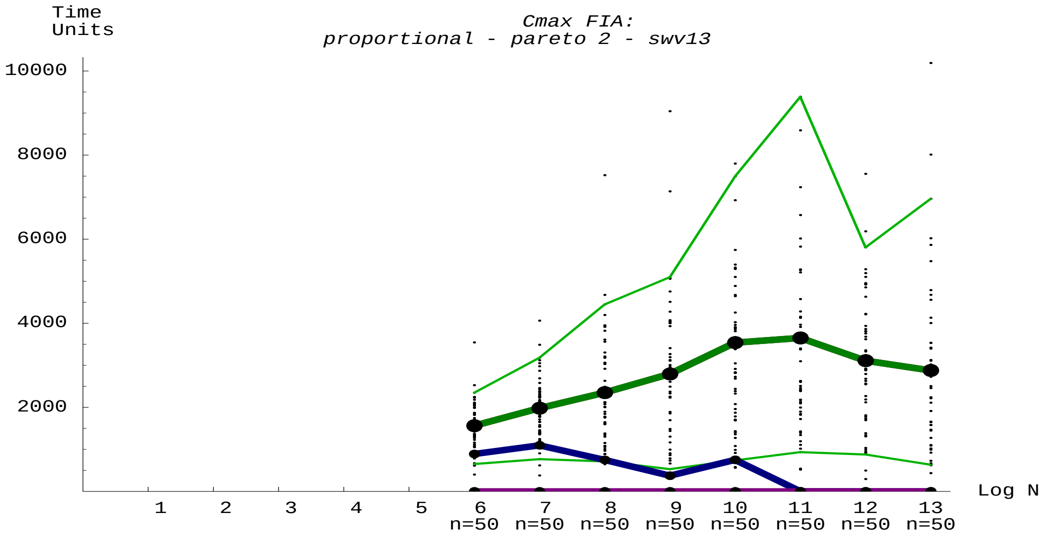


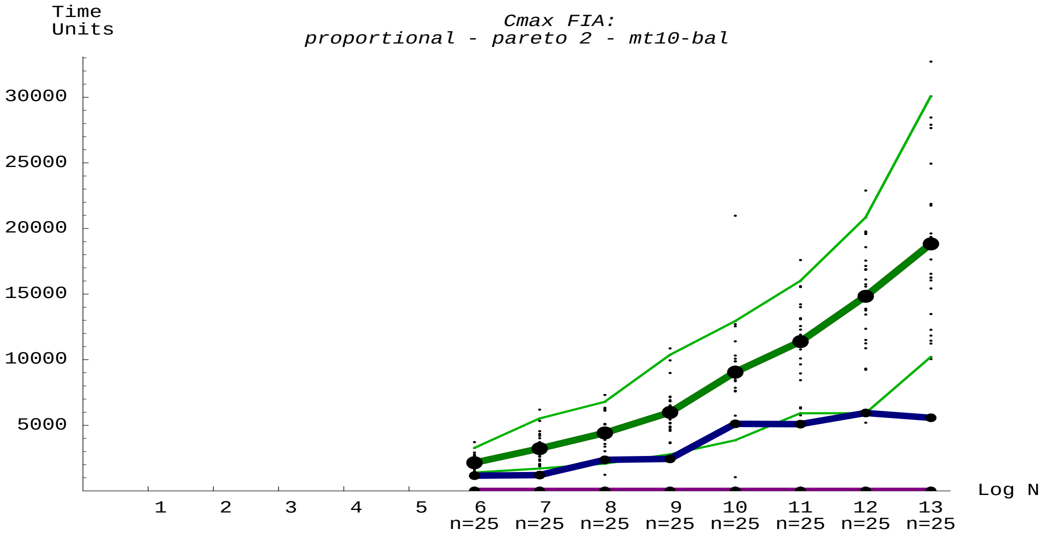
A.2.6 Pareto 2

$C_{\max}$ FIA

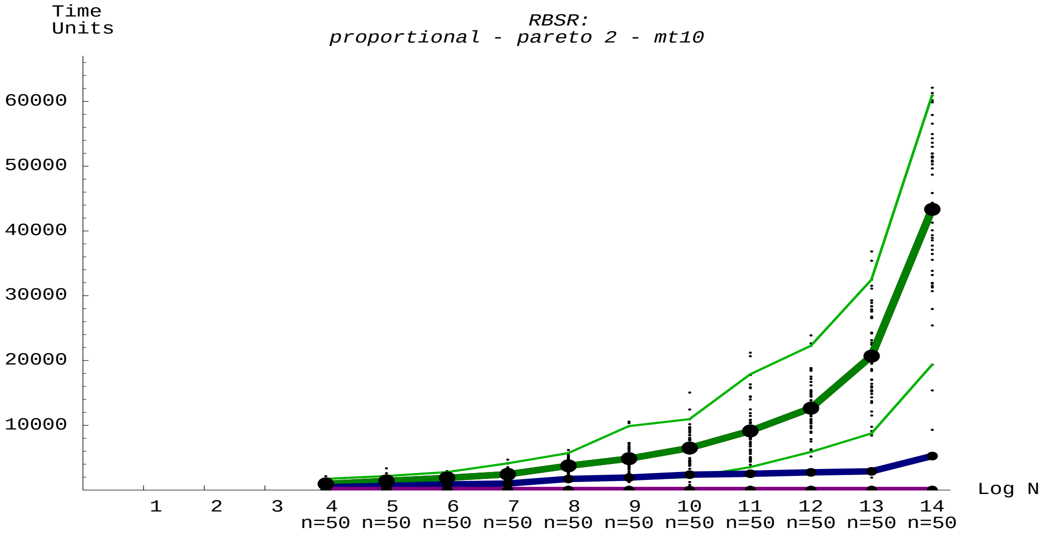


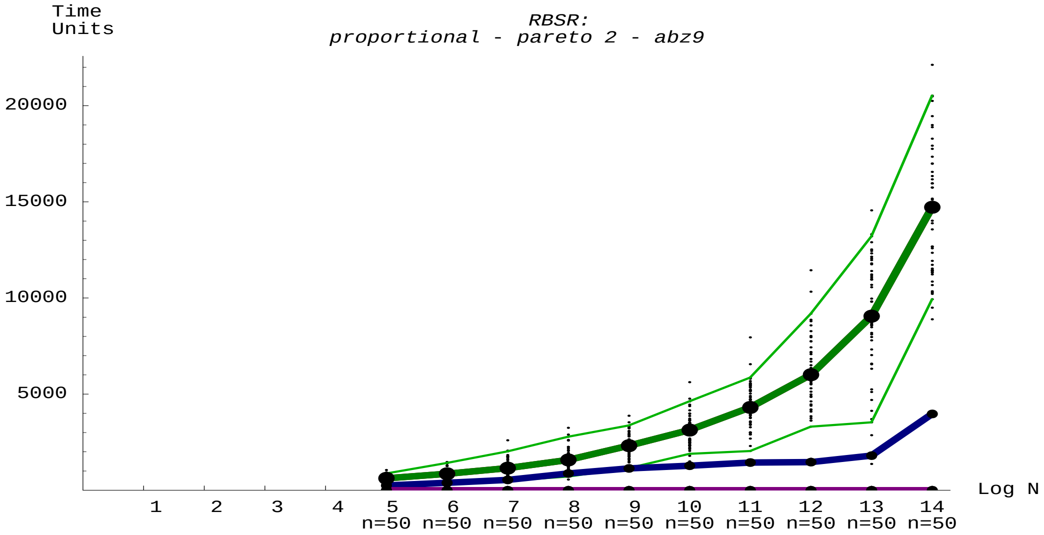

Time
Units
Cmax FIA:
proportional - pareto 2 - abz9
3000
2500
2000
1500
1000
500
Log N
1
2
3
4
5
6
7
8
9
10
11
12
13
n=50
n=50
n=50
n=50
n=50
n=50
n=50
n=50
n=50

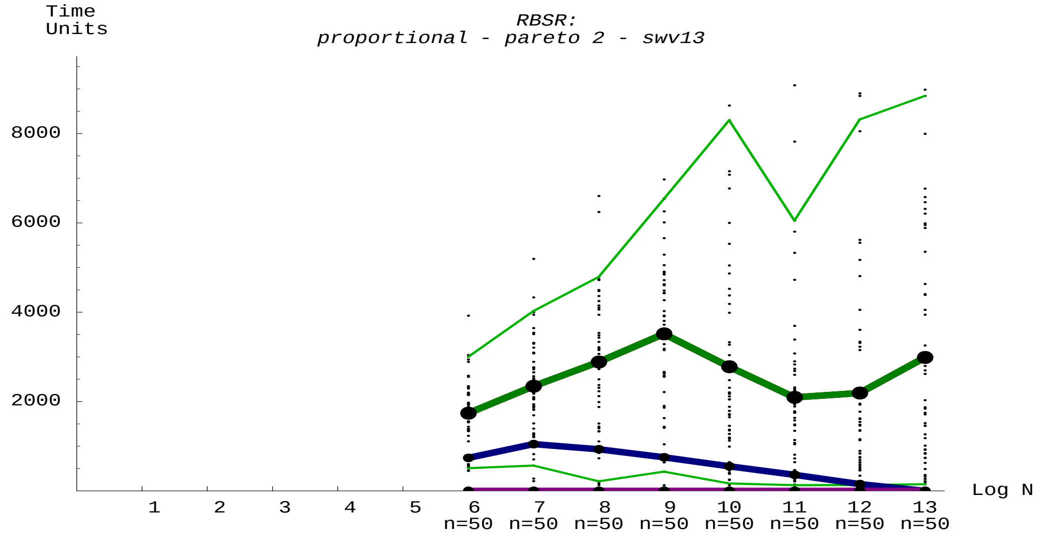

Time
Units
Cmax FIA:
proportional - pareto 2 - swv13
10000
8000
6000
4000
2000
Log N
1
2
3
4
5
6
7
8
9
10
11
12
13
n=50
n=50
n=50
n=50
n=50
n=50
n=50
n=50

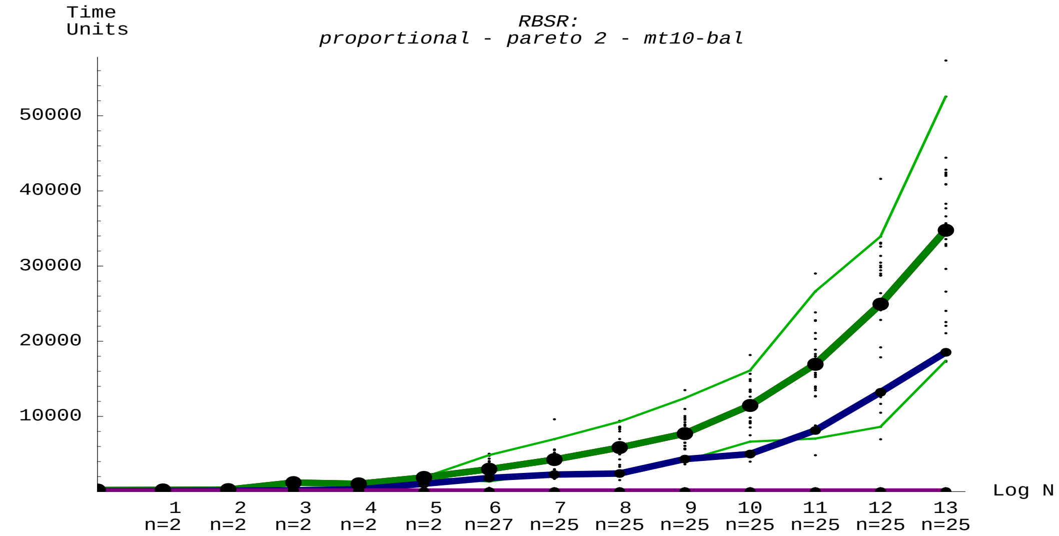

Time
Units
Cmax FIA:
proportional - pareto 2 - mt10-bal
30000
25000
20000
15000
10000
5000
Log N
1
2
3
4
5
6
7
8
9
10
11
12
13
n=25
n=25
n=25
n=25
n=25
n=25
n=25
n=25

RBSR

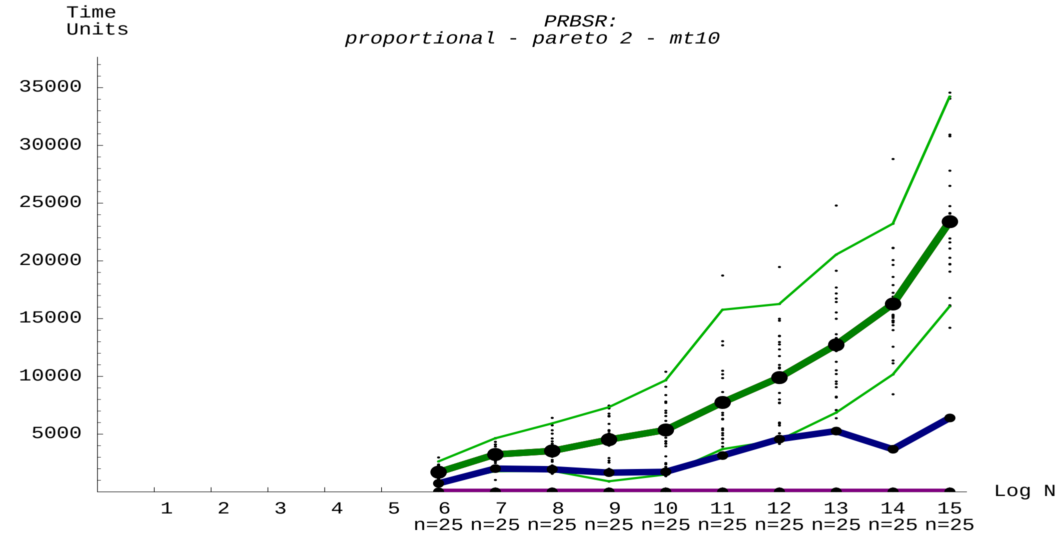

Time
Units
RBSR:
proportional - pareto 2 - mt10
60000
50000
40000
30000
20000
10000
Log N
1
2
3
4
5
6
7
8
9
10
11
12
13
14
n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50


Time
Units
RBSR:
proportional - pareto 2 - abz9
20000
15000
10000
5000
Log N
1
2
3
4
5
6
7
8
9
10
11
12
13
14
n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50

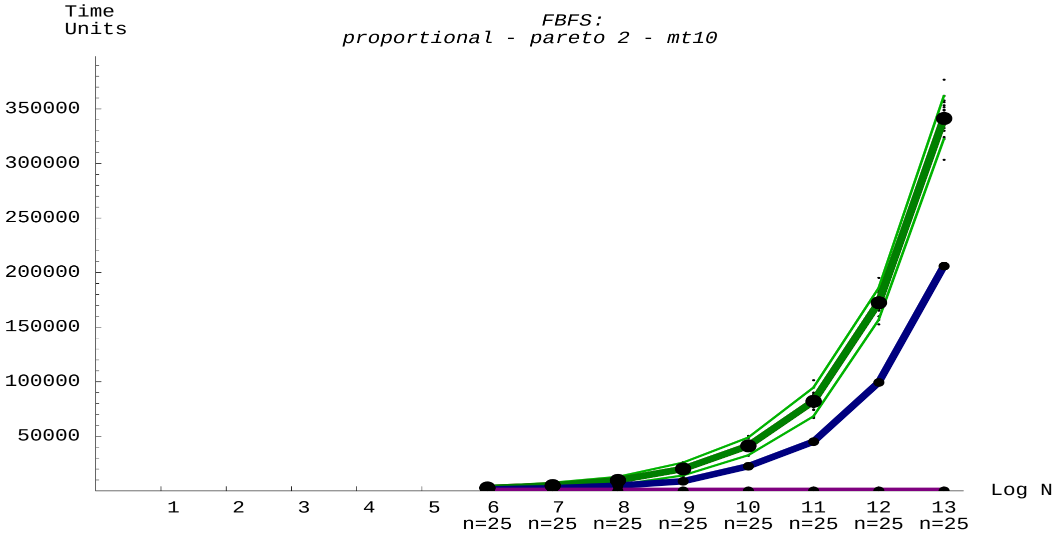

Time
Units
RBSR:
proportional - pareto 2 - swv13
8000
6000
4000
2000
Log N
1
2
3
4
5
6
7
8
9
10
11
12
13
n=50 n=50 n=50 n=50 n=50 n=50 n=50 n=50


Time
Units
RBSR:
proportional - pareto 2 - mt10-bal
50000
40000
30000
20000
10000
Log N
1 n=2
2 n=2
3 n=2
4 n=2
5 n=2
6 n=27
7 n=25
8 n=25
9 n=25
10 n=25
11 n=25
12 n=25
13 n=25

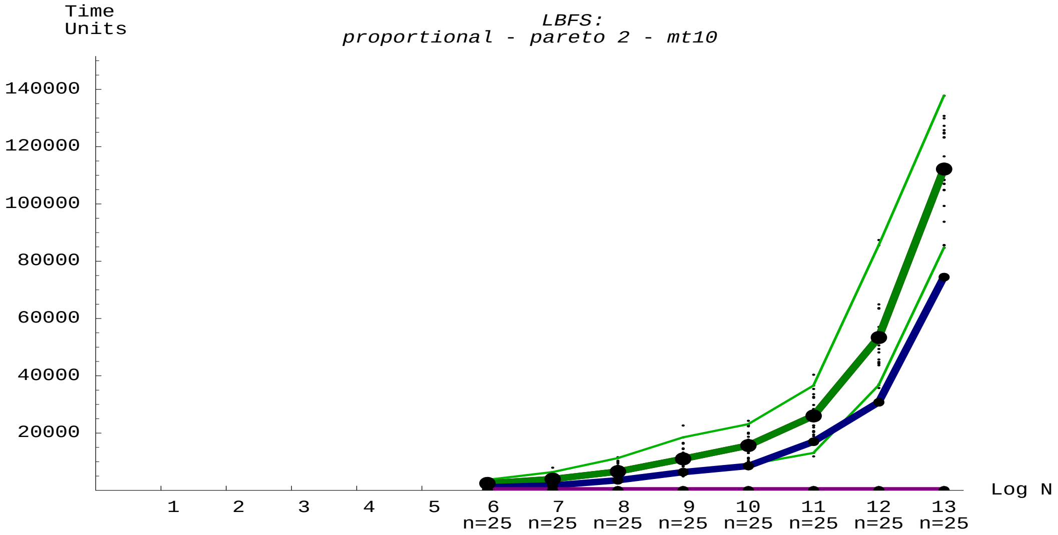
PRBSR


Time
Units
PRBSR:
proportional - pareto 2 - mt10
35000
30000
25000
20000
15000
10000
5000
Log N
1
2
3
4
5
6 n=25
7 n=25
8 n=25
9 n=25
10 n=25
11 n=25
12 n=25
13 n=25
14 n=25
15 n=25

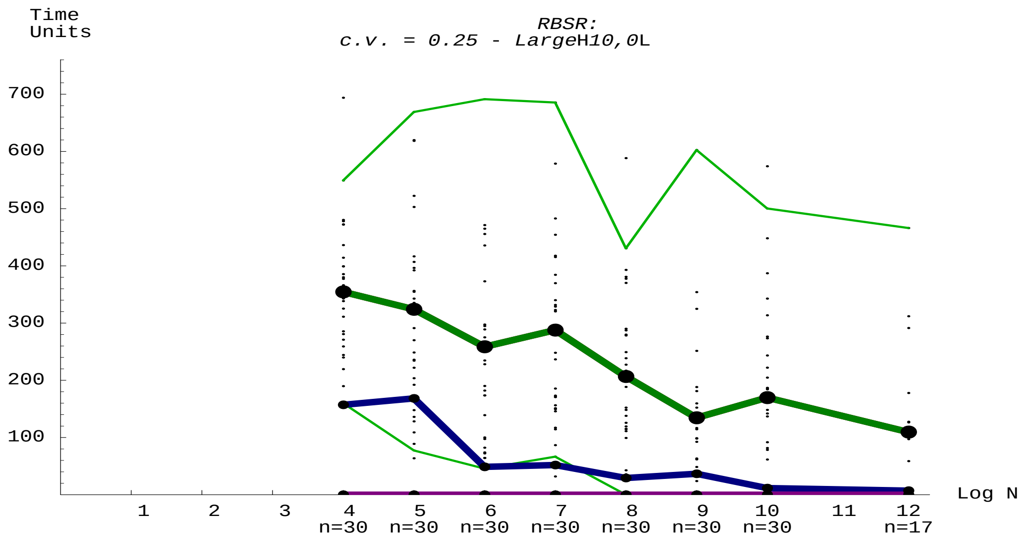
FBFS

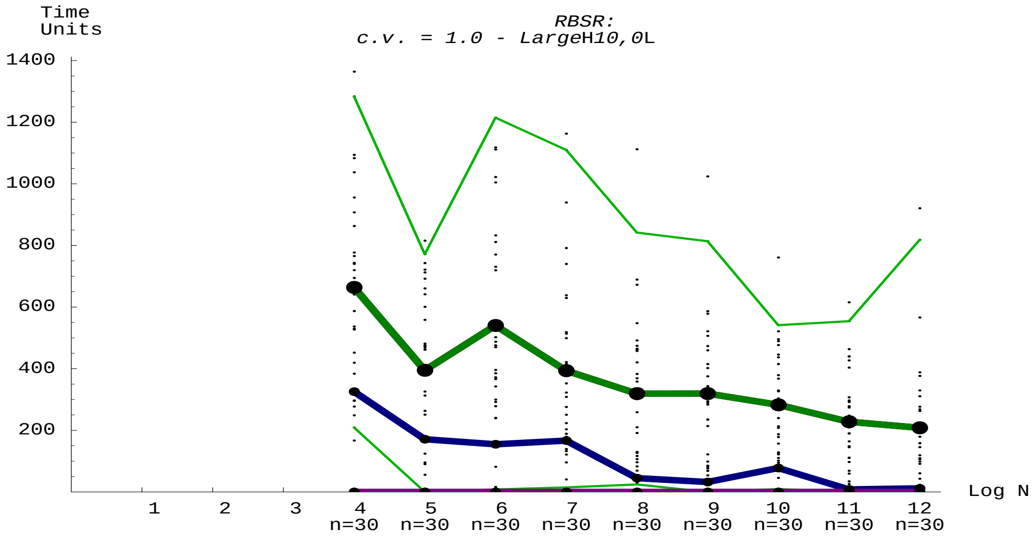

Time
Units
FBFS:
proportional - pareto 2 - mt10
350000
300000
250000
200000
150000
100000
50000
Log N
1
2
3
4
5
6
n=25
7
n=25
8
n=25
9
n=25
10
n=25
11
n=25
12
n=25
13
n=25

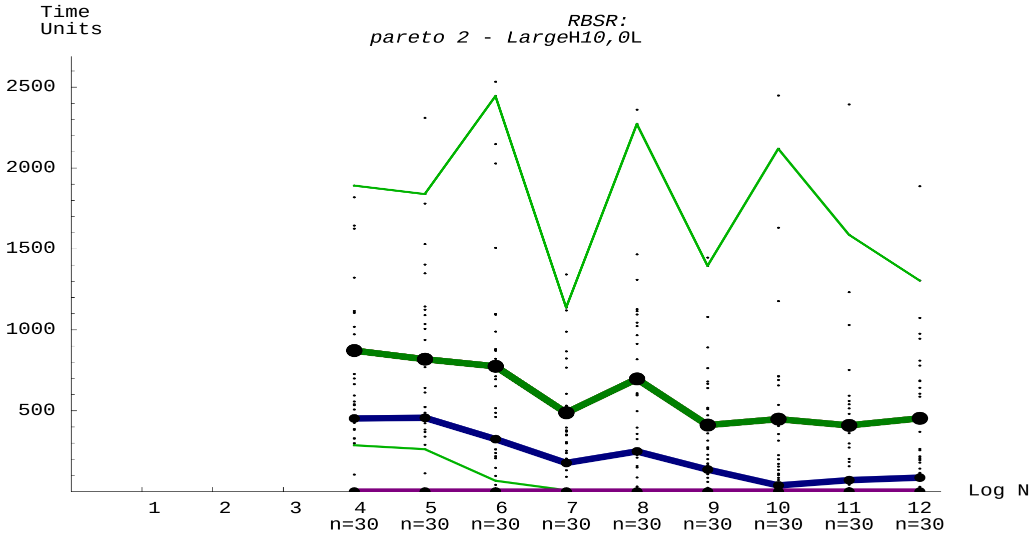
LBFS


Time
Units
LBFS:
proportional - pareto 2 - mt10
140000
120000
100000
80000
60000
40000
20000
Log N
1
2
3
4
5
6
n=25
7
n=25
8
n=25
9
n=25
10
n=25
11
n=25
12
n=25
13
n=25

A.3 The $R \approx J$ Case with RJSR







Appendix B

Simulation Details

This appendix lists the technical details regarding the simulation experiments. Appendix Section B.1 details the way in which the R.V.'s were generated. Appendix Section B.2 lists the proportions that were used for the $R \ll J$ proportional problem instances. Appendix Section B.3 explains how random $R \approx J$ topologies were created.

B.1 Generation of Random Variables

There are two ways in which a random $\wp$ may be generated: (1) Generate all of the random processing times before the beginning of the simulation. (2) Generate the random processing times during the course of the simulation. We used the latter method. At each time that the simulation kernel was interested in the processing time of a job on a machine, a random variable was generated. The generation routines were simple to implement, they are based on the random number generator supplied in the JAVA environment's `java.util` package: `class Random`. `class Random` uses a 48-bit seed which is modified using a linear congruential formula[1], for more information see the JAVA API. A good overview of R.V. generation and transformation may be found in [8].

Since a meaned topology specifies the mean processing time, generation of the random variables requires that the R.V.'s come from the desired distribution around the specified mean. We now briefly describe how we generated the C.V.= 0.25, C.V.= 1.0, Weibull 1/2, Pareto 3 and Pareto 2 R.V's. For each R.V. we describe how it is generated given a mean μ ($\mu > 0$).

We generated the C.V.= 0.25 R.V's by transforming that value that is returned by `java.util.Random`'s `nextGaussian()` method. This is a $N(0,1)$ random value (it is generated using the so-called polar method). Since we are interested in an R.V. having a standard deviance of 0.25μ and a mean of μ, we multiply the `nextGaussian()` value by

[1] This is the implementation as of JAVA v1.3.

0.25μ and add μ to it. Note that there is a $\Phi(-4) \approx 10^{-4}$ probability (Φ is the cumulative distribution function of the standard normal R.V.) that this R.V. is negative. When this un-probable event occurs we re-sample another $N(0,1)$ R.V.

We used the exponential distribution to generate C.V.= 1.0 R.V's. This is achieved by applying the inverse probability transform on a uniform R.V. If U is the uniform R.V. generated by the `nextDouble()` method, then $-\mu \log U$ is distributed exponentially with mean μ.

For the generation of the Weibull 1/2 R.V, we first generated an exponential random variable with mean $\sqrt{\mu/2}$. It turns out that the square of this exponential R.V. is a Weibull R.V. with mean μ and a hazard rate proportional to $e^{-1/2}$.

The Pareto 3 and Pareto 2 R.V's were generated from a Pareto distribution having a cumulative distribution function of the form:

$$1 - (\frac{\lambda}{\lambda + x})^{\alpha}.$$

This is a function of x (for all $x > 0$); α and λ are parameters. When $\alpha = k$ (integer), only the first k moments exists. Thus we used $\alpha = 3$ and $\alpha = 2$ for the two types of Pareto R.V's. Given α and μ, we computed λ accordingly. The inverse of the cumulative distribution function is easily calculated and is applied on a uniform random variable.

B.2 Selection of the Number of Jobs on Each Route in the $R \ll J$ Proportional Case

We tested the $R \ll J$ proportional case with the MT10, ABZ9, SWV13 and MT10-bal meaned topologies. These meaned topologies contain 10, 20 and 50 routes respectively. We arbitrarily chose a certain asymmetric distribution of job on routes for each of the problem types. For the MT10 and MT10-bal problems we used this distribution:

0.04	0.04	0.04	0.04	0.04	0.10	0.10	0.1	0.25	0.25

As can be seen, 50% of the jobs are on 2 of the routes and the other 50% are on the remaining 8 routes.

For the ABZ9 problems we used this distribution:

0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.04
0.04	0.04	0.04	0.04	0.06	0.06	0.08	0.10	0.12	0.20

For the SWV13 problems we used this distribution:

0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.01	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02
0.02	0.03	0.03	0.03	0.04	0.04	0.05	0.05	0.08	0.15

B.3 Creation of $R \approx J$ Job Shop Problems

We generated $R \approx J$ problem instances by using the `RLikeNShopData` class. This class generates a problem instance based on the machines means and the parameters I and a (as specified in Section 4.2.2). The generation of a problem instance is simple: First draw the number of machines on each route by using a discrete uniform $[I-a, I+a]$ distribution, then randomly draw a machine for each of the operations (using a uniform distribution). The processing times are all set to the processing means. As the simulation runs, the random number generator of the simulation is used to generate random processing times for each of the operations, based on the machine's mean[2].

[2] In all of the $R \approx J$ tests performed in this study, all of the machine means were equal

Appendix C

User Instructions for the Job Shop Simulator

Below are the instructions for using the JSS. These instructions are taken from the on-line help that is available with the software. In Appendix Section C.1 we present general instructions for using the JSS. In Appendix Section C.2 we present instructions for creating `.jbs` files.

C.1 Using the Job Shop Simulator

The JSS is an interactive simulation program designed to simulate job shop problems in which the number of routes is fixed and the number of jobs on each route is large.

The JSS's workspace is divided into the following areas:

- The Algorithm Area
- The Gantt Chart Area
- The Shop State Area
- The Shop Data Area
- The Animation Area

Note that you may use the sliders that separate these areas to resize, hide or re-show these areas.

To run a simulation, you must first open a `.jbs` file. Use the *File Menu* to open a file (you may look at the help for `.jbs` files to see the specification of the `.jbs` file). After selecting a `.jbs` file, the JSS loads the file and attempts to understand the information written in it. If the information is ill formed, you will be notified. If the `.jbs` does not contain graphic information you will also be notified and the animation will be suppressed during the course of the simulation. If the information is kosher, the job shop is loaded and summarized in the shop data area (bottom left)

After successfully opening a file, you should select an algorithm from the *Algorithm Menu*.

After selecting an algorithm you may set several options from the *Simulation Menu*. These are the R.V. to be used to generate processing times (around the means specified in the `.jbs` file), the number of jobs in each route and the simulation speed. Note when setting the number of jobs on each route you may actually edit the text fields so that there is an uneven number of jobs on the routes (make sure to press enter after editing a text field). Note also that you do not have to set the simulation speed before starting the simulation, this may be done while the simulation is running.

When you are ready, select *Start Simulation* from the *Simulation Menu*. This will generate the gantt and animation of the job shop (only if graphics information is available in the loaded `.jbs` file). If you selected the *User Controlled Algorithm* you may now start to schedule the shop by yourself. If you selected any other algorithm, click the *Go Button* to start and use the *Step* and *Pause* buttons at your will.

At any moment, during or after the simulation, you may select *View Kernel Log* to view messages that are displayed by the simulation kernel and the algorithm. Note that when the simulation is complete, the statistics collector dumps the collected statistics to the kernel log.

Note that machines and routes are labeled using the numbers 1,2,3,.... Jobs are not labeled because the simulation engine treats all jobs on the same route as having the same processing time distribution. Operations are labeled (r,o), where r is the route number and o is the operation number. In the shop state area you may see the current number of jobs in each buffer (each operation), the numbers displayed, take into account any jobs that are currently being processed. In the animation, jobs are the little purple dots, buffers (operations) are the yellow trapezoids, working machines are red circles and idle machines are blue circles. The big triangles are route starts and the big boxes are route ends.

C.2 Creating Your Own .jbs Files

A `.jbs` (Job Shop) file is a file that describes the meaned topology of a high volume job shop problem instance. This file is read by the JSS and used to perform the simulations. The file is to be created by any text editor, it may then be opened by the JSS when run as an application or put on a web server when the JSS is run as an applet. Several example files are available with the installation.

The file format uses the following guidelines:

1. The file is divided into three logical parts: The *shop dimension* part specifies how many machines and how many routes are in the shop, the *routes part* specifies the steps and mean durations of each route and the *graphics part* specifies the information that tells the shop animation how to display the shop (this part is optional).
2. Comments may appear in the file in lines that follow the # character.
3. All numbers may be of a floating point form: `3.14` or an integer form: `354`
4. All directives may be either lowercase, uppercase or a mix.
5. All information should be separated by white space (space,tab or enter).

The Shop Dimension Part

In the shop dimension part (the start of the file) you should specify two integer numbers, the first is the number of machines and the second is the number of routes.

The Routes Part

In the routes part you should enter a number of *route entries* that is the same as the number of routes specified in the shop dimension part. Each route entry should look like this:

```
Route 14
3 5 9 3 8
Means
3.4 3 5 2 5.0
```

The first line of numbers are the machine indexes for route 14, the second line is the processing means for each of the steps. Thus the route entry means that route 14 is composed of 5 steps with the given machines and processing times.

The Graphics Part

This part is optional (it may be omitted). Note that for shops with more than 10 routes or more than 25 machines you may not specify graphics information. There is also a limit of up to 12 buffers (operations) per machine.

The graphics part starts with the word `Graphics`. When the JSS reads the `.jbs` file, it uses the 'Graphics' word to understand that you have put graphic information. After the word 'Graphics', you should put two parts, the *machines part* and the *routes part*. The machines part consist of entries that tell the JSS, where and how to display each machine. The routes part tells the JSS where and how to display the route ends and starts (squares and triangles respectively). Also, the machines part consists of *machine entries*, one for each machine and the routes parts consists of *route entries*, one for each route. You must make sure that the number of entries matches the number of machines and number of routes. Here is an example of a machine entry:

```
Machine 5
3 2
1 4 30
2 5 60
4 8 120
7 3 240
```

This is the entry for machine 5. The first two numbers specify the location of the machine on the graphics grid (a 5 by 5 grid). This should be a coordinate of the type (i,j) i=1,...,5 j=1,...,5. It is your responsibility to make sure that there is no other machine with the same graphics coordinate. (You now see why there is a limit of 25 machines (for graphics info). The next four lines in the machine entry are directives for displaying the buffers in the machine. Each machine may have up to 12 buffers (a buffer is synonymous with an operation). The machine in the example above has the buffers (1,4), (2,5), (4,8) and (7,3). On the screen, the buffers are displayed as yellow trapezoids placed on the circumference of the machines. For each buffer, we specify an angle at which it is to be displayed (the angles are multiples of 30 degrees). Since a circle has 360 degrees, there is a limit of up to 12 buffers per machine (a graphics limit, not a simulation limit). An angle of 30 degrees, means that the buffer is placed at 1 o'clock. An angle of 270 degrees, means that the buffer is placed at 9 o'clock etc ...It is your responsibility to make sure that the shop data, really matches the buffers that you specified (the JSS will not catch this, it will stall if there is a mismatch). Thus in the above example, buffers (1,4), (2,5), (4,8) and (7,3) should be the buffers that are specified in the routes (1, 2, 4 and 7) to be on machine 5.

After specifying machine entries for each of the machines, you should specify route entries. Each route is characterized by an start (triangle) and end (square). The route tips (start and ends) are to be placed on the perimeter of the 5 by 5 grid that is allocated for the machines. Excluding the corners this leaves room for 20 route ends (a total of 10 routes). Let's look at an example route entry:

```
Route 8
NORTH 3
SOUTH 5
```

This entry is for route 8. It says that the start should be on the north side at location 3 (index of 1 to 5, west to east and north to south). The end in the example above is place on the south side on location 5. You may have put `EAST` or `WEST` instead of `NORTH` and `SOUTH`, or any combination. Just make sure that two route tips are not assigned to the same spot.

Appendix D

Source Code Listing

The next pages include a partial source code listing for the Job Shop Simulation Project (version 1.2). The full source code is available at this Internet site:

`http://rstat.haifa.ac.il/~yonin/thesis/jobshopsim/shopsim.html`

Note that JAVA offers the Javadoc utility. This utility allows generation of easy to read documentation for the code. Javadoc generated documentation for the code may be down-loaded from the Internet site.

The source code spans around 100 files (only a small sample is presented below). In the listing below, each file starts on a new page. The page numbering restarts with each file. Note that the column of numbers to the left of the code are not part of the JAVA language.