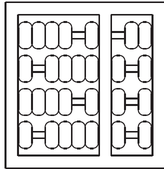

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Using Genetic Algorithms for the Dynamic Job Shop Scheduling Problem with Processing Time Uncertainties

Gregório Baggio Tramontina *Jacques Wainer*

Technical Report - IC-05-002 - Relatório Técnico

February - 2005 - Fevereiro

The contents of this report are the sole responsibility of the authors.
O conteúdo do presente relatório é de única responsabilidade dos autores.

Using Genetic Algorithms for the Dynamic Job Shop Scheduling Problem with Processing Time Uncertainties

Gregório Baggio Tramontina* Jacques Wainer

Abstract

A possible solution to the job shop scheduling problem is using genetic algorithms. Although research on the topic can be considered evolved, it mainly concentrates on deterministic static problems. This work reports an experience on using genetic algorithms with the dynamic job shop problem with processing time uncertainties, based on two approaches: the guess and solve, and the decomposition of dynamic job shops into static instances. The guess and solve consists of making a controlled guess on the processing time of the jobs and solving the resulting deterministic scheduling problem with a suitable technique, in this report the genetic algorithms, and the decomposition treats the dynamic job shop as a series of static problems. Processing time uncertainties prevent the direct use of the decomposition approach, so an adjustment to this technique is also proposed. Simulation results show that the guess and solve with genetic algorithms together with the proposed adjustment is an adequate way to handle the dynamic job shop with processing time uncertainties.

1 Introduction

One of the most popular models in scheduling theory is that of the *job shop*. A possible solution to it comes from the *genetic algorithms*. Results from many researchers, for example [1, 2, 4, 5, 10] show that this approach is adequate, generating fairly good results against some of the most used scheduling techniques.

Much of the job shop research with genetic algorithms has been concentrated in static deterministic scheduling contexts. In these contexts, all information about the problem is at hand at the time the solving algorithm is executed, and there is no change to the final schedule. But in a shop floor the job releases are typically unforeseen events and, consequently, neither the release times of jobs nor their specific processing requirements are known in advance [2]. Thus many researchers have also developed good solutions to the dynamic job shop problem. But they mainly concentrate in those problems where the processing times of the jobs are exactly known. This is also not so accurate, as for example in workflow problems, even these processing times are uncertain; there is at best a (possibly) good approximation to the time a workflow activity takes to be executed. It also clearly places the problem in the stochastic domain.

*Partially supported by the Fundação de Amparo à Pesquisa do Estado de São Paulo - FAPESP - Brazil.

This report presents the authors' experience on applying genetic algorithms to the job shop in a dynamic context and with processing time uncertainties. To solve this dynamic problem a decomposition of its dynamic aspect as a series of static scheduling problems is done as described by Raman et al. in [11]. Each of these static problems face the fact that the processing times are uncertain, so a convenient abstraction to model these uncertainties is described: the *guess and solve* approach. It consists of making a controlled guess on the jobs' processing times and solving the resulting deterministic problem with a suitable technique, which in this report is a genetic algorithm.

The processing time uncertainties prevent the direct use of the decomposition approach, so an adjustment which allows the application of the decomposition's concepts to the guess and solve, is presented. Careful simulation of well known dispatching rules and the guess and solve is conducted, and its results shown that the guess and solve with a genetic algorithm, together with this adjustment, is a valid approach to minimize the mean percentage of late jobs in a scheduling scenario.

2 The Job Shop Scheduling Problem

The job shop scheduling problem (JSSP) is defined as a set J of n jobs that have to be processed in one or more machines of a set M which contains m machines. Each job j has its own route to follow through the machine set, so for j there is a technological order μ_j which represents the sequence of machines it will follow. The total number of processing steps a job has to complete is represented as m_j. Each k-th processing step of a job in the system, $1 \leq k \leq m_j$, is defined as an operation $o_{j(k)}$.

The technological order μ_j can be considered as a function that maps the operations a job will execute to the actual machines where these operations take place, and so $\mu_j(k) = i$ means that operation $o_{j(k)}$ will be executed in machine i, $1 \leq i \leq m$. For example, $\mu_1(3) = 2$ means that job 1 will execute operation $o_{1(3)}$ (its third step) in machine 2.

Table 1 shows an example of a JSSP. To this problem, there is a machine set $M = \{1, 2, 3\}$ and a job set $J = \{1, 2, 3\}$. The table shows, for each job, in which machines their operations will take place. While job 1 performs three operations in machines 1, 3 and 2, job 2 performs only two operations in machines 2 and 1 and job 3 has only one operation in machine 3.

Operation	Job 1	Job 2	Job 3
1	1	2	3
2	3	1	-
3	2	-	-

Table 1: Example of a JSSP

A schedule of a set of jobs in the JSSP can be seen as a table of starting times $t_{j(k)}$ for the operations $o_{j(k)}$ with respect to the technological machine orders of jobs [2]. Each job j has a processing time $p_{i,j}$ in a machine i from M. In a dynamic context, job j also has a

release date r_j, which is the point in time where it arrives at the system. The completion time of an operation $o_{j(k)}$ is given by its starting time plus its processing time ($t_{j(k)} + p_{i,j}$, where $\mu_j(k) = i$).

According to Jain and Meeran in [6], although there are specific cases of the JSSP which can be solved optimally in polynomial time, there are $(n!)^m$ possible solutions to it, and even though many of these solutions will not be feasible due to precedence and disjunctive constraints, complete enumeration of all the feasible sequences to identify the optimal one is not practical. As a result of this intractability the JSSP is considered to belong to the class of decision problems which are NP. And, for those instances that can be solved efficiently, even a slight modification in a problem's definition may transform it in NP-hard or strongly NP-hard. This is one of the reasons why solutions based in local search mechanisms, including genetic algorithms, have received much attention recently.

3 Applying a Genetic Algorithm to the JSSP

Since Davis started to use genetic algorithms (GAs) to solve scheduling problems in 1985 plentiful research has been developed in this area [1]. Many different situations have been studied, both in static and dynamic contexts. But the authors noticed that these studies were mainly concentrated on situations where the jobs' processing times are known in advance. In this section we will be discussing our experience on applying a GA to the JSSP when all we have is a (possibly) good approximation to these processing times.

3.1 The JSSP Representation

The problem representation is an important decision when working with genetic algorithms When the representation is suitable for the problem at hand, it contributes to better results from the GA.

The chosen representation here is one presented by Bierwirth in [1, 2]. In this representation a possible schedule to the problem is a permutation of the operations of all jobs in the system. For the example given in table 1 a possible representation should be the permutation $\{o_{1(1)}, o_{1(2)}, o_{1(3)}, o_{2(1)}, o_{2(2)}, o_{3(1)}\}$.

This representation scheme cannot be used alone because it does not have enough information to stablish the order within which these operations will be carried out in each machine. All we have in a operations permutation is a relative order among them. That is why Bierwirth gives an algorithm to translate such permutations into feasible schedules containing the operations' starting times in each machine. This translation algorithm is given next, as Bierwirth describes it in [1].

1. Build the set of all beginning operations, $A = \{o_{j(1)} | 1 \leq j \leq n\}$.
2. Determine the earliest possible starting time t' of all operations in A, $t' \leq t_{j(k)}$ for all $o_{j(k)} \in A$.
3. Build set B from all operations in A which have their earliest possible starting time at t', $B = \{o_{j(k)} \in A | t_{j(k)} = t'\}$.

4. Select operation $o^*_{j(k)}$ from B which occurs leftmost in the permutation.
5. Delete operation $o^*_{j(k)}$ from A, $A = A\ \{o^*_{j(k)}\}$.
6. If $o^*_{j(k+1)}$ exists, insert it into A, $A = A \cup \{o^*_{j(k+1)}\}$.
7. If $A \neq \emptyset$ go to step 2, else terminate.

The algorithm guarantees the generation of feasible schedules, that is, no matter what permutation is used as input, it always gives a correct schedule as output. This eliminates a further feasibility check step for the generated schedules. It also generates non-delay schedules [1], that is, schedules where no machine is kept idle as long as there are jobs to be processed in it.

3.2 The Genetic Algorithm

Plentiful research has been conducted in the genetic algorithms field, so many adaptations, differentiations or specifications of the general GA have been proposed and used successfully in many sorts of problems. But to the experiments shown here it was preferred to use the general GA so these modifications would not alter its basic behavior, which is exactly the analysis subject.

The GA used here is a fusion of the general genetic algorithm given by Mitchell in [8] and the mutation scheme proposed by França et al. in [5]. While the crossover operation is done like Mitchell proposes, mutation is performed like the implementation of França.

3.2.1 Generation of the Initial Population

A simple mechanism is used to generate the initial population for the algorithm. Based on a first permutation representation (containing all the operations to be scheduled) extracted from the system, another one is generated randomly selecting two positions of the first representation and swapping their operations. In the next step, there will be two population individuals, so one of them is randomly selected to serve as the base representation from which to extract another one. In the next step, there will be three individuals to be selected, and this process goes on until all the initial population is generated. Each representation is implemented as a vector with indexes ranging from 0 to $P - 1$, where P is the number of operations the representation codifies. The algorithm used to the generation process is given next.

1. Build vector $Pop[0..N-1]$ which contains the first permutation representation pr_0 extracted from the system as its first element, $Pop[0] = pr_0$ (N is the size of the population).
2. Randomly select an index ind between 0 and the number of valid elements Pop already contains minus 1, $ind = \text{RandomIndex}(|Pop| - 1)$.
3. Randomly select two indexes ind_1 and ind_2 from the element $Pop[ind]$, $ind_1 = \text{RandomIndex}(|Pop[ind]| - 1), ind_2 = \text{RandomIndex}(|Pop[ind]| - 1)$.

4. Swap the elements in positions ind_1 and ind_2 of $Pop[ind]$, Swap($Pop[ind][ind_1], Pop[ind][ind_2]$).

5. If enough individuals to fulfill the initial population have been generated, stop, else go to step 2.

3.2.2 Crossover

The crossover operator used was the well known *Order Based Crossover*, or OX. Like in many other operators, an offspring (one new chromosome in this case) is generated based on two parent chromosomes, selected according to their fitness function: the higher the chromosome's fitness, the higher its probability of being chosen will be.

Let $pr_1[0..P-1]$ and $pr_2[0..P-1]$ be vectors containing the representations of the chosen parent chromosomes, and let $o[0..P-1]$ be the offspring generated by the crossover operator. Let also Pop_{n} be the new population being generated by the crossover-mutation process. In the OX operator, the following algorithm is used to generate the offspring.

1. Randomly choose two indexes i_1 and i_2 in the interval $[0..P-1]$, $i_1 = \mathrm{RandomSelect}([0..P-1])$, $i_2 = \mathrm{RandomSelect}([0..P-1])$.

2. Copy the portion of parent pr_1 ranging from i_1 to i_2 to the offspring, in the same position the elements appear in the parent, $o[k] = pr_1[k]$ for $i_1 \leq k \leq i_2$.

3. Scan parent pr_2 from left to right, copying all elements that do not appear in the already copied portion of parent pr_1 to the unfilled portions of the offspring, also from left to right: for each element $pr_2[k]$, $0 \leq k \leq P-1$, if $pr_2[k] \neq o[s]$, $\forall i_1 \leq s \leq i_2$, $o[l] = pr_2[k]$, $(0 \leq l \leq i_1 - 1)$ or $(i_2 + 1 \leq l \leq P-1)$.

4. Insert offspring in the new population Pop_{n}, $Pop_{\mathrm{n}} = Pop_{\mathrm{n}} \cup \{o\}$.

Figure 1 illustrates steps 2 and 3 of the crossover algorithm. In the example, two population individuals, which are permutations of operations A, B, C, D, E, F, G, H and I, are chosen to be the parents. Considering the representation positions starting from 0, $i_1 = 3$ and $i_2 = 6$. F H I E, the portion of parent 1 between i_1 and i_2, is copied to the offspring. Then parent 2 is scanned from left to right and the empty positions of the offspring are filled in with elements from parent 2 which are not in the portion of the offspring copied from parent 1.

3.2.3 Mutation

Mutation is done like the implementation of França et al. in [5]. Instead of mutating a certain percentage of the population, we use a mutation probability mut_{p}, that is, every representation will be mutated with probability mut_{p} $(0 \leq mut_{\mathrm{p}} \leq 1)$. This saves time when implementing the algorithm and does not severely change the final results.

In the mutation scheme adopted here, as soon as an offspring o from the crossover operation is generated, it is probabilistically mutated according to the value of mut_{p}, i.e., a random number between 0 and 1 is chosen and if this number is smaller than or equal to mut_{p}, then o is mutated swapping two of its elements, also randomly selected.

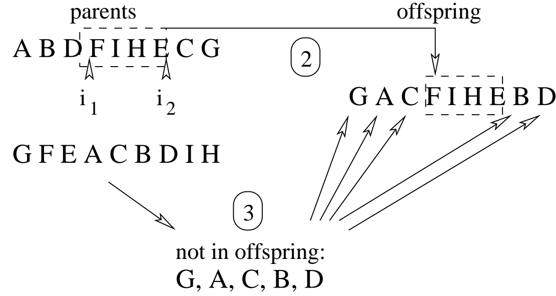

Figure 1: Example of the OX crossover operator

3.2.4 Fitness Function

Since the GA will be used to reduce the percentage of late jobs, the chosen fitness function was one that prioritized schedules with as few late jobs as possible. So the fitness of a representation h_k from the GA's population is given by

$$\text{fitness}(h_k) = \frac{1}{1 + \text{nlate}(h_k)}$$

where $\text{nlate}(h_k)$ is the number of late jobs in representation h_k.

3.3 Dispatching Rules

Three well known dispatching rules are used in order to provide a comparison basis to our GA approach. They are:

- FIFO (First In First Out): the jobs are executed in the order they arrive at the machines.
- SIRO (Service In Random Order): jobs are randomly chosen to be executed.
- EDD (Earliest Due Date): jobs with closer due dates are processed first.

Dispatching rules are known to give relatively good results and also being computationally cheap. FIFO and SIRO are very common rules because they are easy to use and implement, not needing any special information from a job. The EDD rule is known to be effective in minimizing tardiness related metrics, so it was also chosen.

3.4 Dynamic Scheduling Using a GA

The problem representation mentioned in section 3.1 can be perfectly used in a static scheduling context, in which the jobs are all available to be scheduled in advance and this schedule will never change during the jobs' processing. But in a dynamic context jobs are not known in advance anymore; they arrive at specific, asynchronous points in time instead,

called the jobs' *release dates* r_j, and only after these release dates the system becomes aware of the new jobs. And including these new jobs in the schedule without some rescheduling may deteriorate the system's performance.

A solution in this case is to decompose the dynamic context in many static ones, and schedule each of these static contexts, that is, every time a new job arrives, all jobs that are already in the system plus the new job will be considered a static scheduling context, will be scheduled, and this schedule will be implemented until another new job arrives, when the process starts again. Such decomposition makes the dynamic context to be considered as a series of static ones.

This approach is presented by Raman et al. in [11] and is used by Bierwirth in [2]. In such approach, a first static scheduling problem P_0 is considered, which is composed of all operations that are available at a time t_0. Problem P_0 is solved as a static scheduling context and this solution is given as a table of starting times $t_{j(k)}$ of operations $o_{j(k)}$.

In this scheduling operation, the earliest possible processing time of an operation $o_{j(k)}$ is given by the maximum between the completion times of the operation that precedes $o_{j(k)}$ in the technological order of job j, denoted by $o_{j(k-1)}$, and the operation $o_{h(l)}$ that precedes the execution of $o_{j(k)}$ in the same machine. This situation can be expressed mathematically by the formula

$$t_{j(k)} = \max\{t_{j(k-1)} + p_{\mu_j(k-1),j}, t_{h(l)} + p_{\mu_h(l),l}\} \tag{1}$$

If operation $o_{h(l)}$ does not exist, its corresponding term in the equation is set to 0. If $o_{j(k)}$ is in fact the first operation of a job ($o_{j(1)}$), then the release date of job j is considered, and the earliest possible starting time of this operation is now calculated with the formula

$$t_{j(1)} = \max\{r_j, t_{h(l)} + p_{\mu_h(l),l}\} \tag{2}$$

At this point the already scheduled operations may be implemented in the system.

When a new job arrives at a time t_1, a problem P_1 is generated removing all operations from P_0 that have their starting times $t_{j(k)} < t_1$ and have also completed their execution, that is, $t_{j(k)} + p_{\mu_j,j} \leq t_1$, since they have already left the system and cannot take part on a subsequent scheduling operation. Second, we must recalculate the release dates of the jobs which had operations removed from P_0 but have another ones yet to be scheduled. This recalculation is done by the formula

$$r_j = \max_{1 \leq k \leq m_j} \left\{ t_{j(k)} + p_{\mu_j(k)j} | t_{j(k)} < t_1 \right\}$$

where m_j represents the number of operations still remaining to be processed for job j.

A third step is to identify the operations that have started before t_1 but have not been completed until it was reached. This means that these operations are in a situation where $t_{j(k)} < t_1 < t_{j(k)} + p_{\mu_j(k),j}$. In this case, a machine i, which is executing one of these operations, will not be available at t_1, so an initial setup time is considered to this machine. This setup time is calculated with the formula

$$s_i = \max \left\{ \max_{1 \leq j \leq n} \left\{ t_{j(k)} + p_{\mu_j(k),j} | t_{j(k)} < t_1 \right\}, t_1 \right\}$$

Notice that, in problem P_0, this setup time is 0 for all machines, so they are all available in time t_0 and the system does not face this problem until subsequent job releases happen.

Finally, all the jobs released in time t_1 are added to problem P_1. At this point problem $P1$ forms a new static scheduling problem and can therefore be rescheduled. Now machine setup times are taken into account and if an operation $o_{j(k)}$ is the first to be executed by a machine i, then its earliest possible starting time is calculated by the formula

$$t_{j(k)} = \max\{t_{j(k-1)} + p_{\mu_j(k-1),j}, s_i\}$$

If an operation is the first to be processed by machine i and is also the job's beginning operation in P_1 (not necessarily the first operation of the job, but its first operation in the current problem), its earliest possible starting time is calculated with the formula

$$t_{j(k)} = \max\{r_j, s_i\}$$

The other operations have their earliest possible starting times calculated according to equations (1) and (2). Notice that all these calculations are used in step 2 of the translation algorithm shown in section 3.1, when it has to determine the earliest possible starting times of the jobs.

Until further jobs are released in a time t_2, all operations scheduled before $t2$ can be implemented. Repeating this procedure after each new job release, the production program is rescheduled periodically in a rolling time basis [2].

3.5 The Guess and Solve Approach

The decomposition described in the previous section assumes that the jobs' processing times are deterministic. But processing time uncertaintis are a very important characteristic of some situations modeled as JSSPs. Particularly in the case of this report the motivation to using a JSSP modeling came from a workflow system, where there is at best a (possibly) good approximation to its activities' processing times.

These processing time uncertainties are modeled with what the authors call the *guess and solve* approach. It consists of two phases: making a controlled guess on the jobs' processing times, to bring the problem from the stochastic to the deterministic domain (the *guess* phase), and then solving the resulting deterministic problem with a suitable scheduling technique (the *solve* phase).

The guess phase assumes that the system has a *guesser* capable of providing a prediction $p'_{i,j}$ of the real processing times $p_{i,j}$, and this prediction is wrong with a maximum error f_{p}. This means that the property

$$f_{\text{p}} \geq \left| \frac{p'_{i,j} - p_{i,j}}{p_{i,j}} \right|$$

holds for every prediction.

The guesser is an adequate abstraction to a number of prediction techniques that can be used, such as relying on someone's experience, using machine learning [8], or statistical techniques based on a number of past cases.

The solve phase consists of using a suitable scheduling technique to solve the resulting deterministic JSSP. Many of such techniques have been proposed, and examples can be taken from [6], like branch and bound algorithms, simulated annealing, neural networks, genetic algorithms, tabu search and even some efficient algorithms specially developed to specific JSSP instances. This report uses the genetic algorithms as a possible solution strategy to the JSSP.

3.6 Processing Time Uncertainties: Adjusting the Temporal Decomposition

Once the predictions are the only available data about the jobs' processing times, all scheduling operations are made based on them. But when the scheduling is implemented the real processing times are brought up, so some inconsistent situations may occur. For example, a job j is supposed start processing in machine i in time t_0, stop executing in this machine in time t_1 and immediately start processing in machine l. But the guesser made a prediction $p'_{i,j} < p_{i,j}$. Since the schedule was calculated based on the prediction, the job is expected to last $t_0 + p'_{i,j} = t_1$ in machine i. But its real processing time forces the job to spend $t_0 + p_{i,j} > t_1$, thus leading to a situation where the job must start executing in machine l but has not left machine i yet.

To solve this an adaptation to the decomposition technique is proposed. It consists of using a new step after the translation of a JSSP representation. After this translation has occurred, a table containing all the job's starting times is available. This table is used to create what we call the *order lists* for each machine in the system. These lists represent the exact order the jobs will be processed in the machines.

When a job arrives at a machine, it is put in the machine's queue. Whenever the machine becomes free it searches in its queue for the job whose number is the first in its order list. If it succeeds in this operation, the job is processed; if not, the machine waits for the right job to arrive. If the job arrives and the machine is already idle, it searches its order list to check whether this job is the one it is waiting for; if so, the job is also processed, else it is put in the machine's queue. This modification, although simple, is very convenient because it frees the system from having to recalculate the starting times every time an inconsistency is discovered, and is also effective as we shall see later.

Generating these lists is simple provided that we have already generated the starting times table. It consists of verifying all table positions (remember that each position corresponds to an operation) and using the insertion sort algorithm to put the operations into the order list of the correct machines. The operations are inserted in these lists in an increasing order of their starting times.

4 Problem Definition

The problem studied here is also motivated by a workflow situation the authors have faced. Is is a relatively simple model but contains some important workflow characteristics. It is the modeling of the mentioned workflow situation as a JSSP.

The scenario description is as follows:

- It is composed of 4 machines.
- Each job has its own machine route to follow, and there are two possible routes for these jobs: the sequence of machines (1,2,4) and (1,3,4).
- The real processing times of jobs are stochastic, taken from a uniform distribution over two intervals: for machines 1, 2 and 4, $p_{i,j} \in [12, 28]$; for machine 3, $p_{i,j} \in [5, 5]$. This creates some difference between the time jobs spend in machine 2 and 3 and arrive at machine 4.

The jobs' release dates are set so a certain machine utilization ratio is achieved (this will receive more attention later). Due dates are set multiplying the total processing time of a job by a constant (the *allowance factor*) and adding its release date to this value. They are calculated by

$$d_j = r_j + A \sum_{i=1}^{m} p_{i,j}$$

where A is the allowance factor. The amount of time within which a job can be completed without being late, calculated by $d_j - r_j$, is called the job's *allowance*.

4.1 Workload Generation

The *workload* is considered to be a crucial aspect of a system's performance [2]. Also called *work in process* (WIP), it is defined as the number of tasks waiting to be processed in the system [2, 7, 9]. It is affected by two factors: the arrival and the completion of jobs in the system.

To control the system's workload, we use Mattlefield and Bierwirth's [2, 3] modeling of the arrival of new jobs to the system. In this model, the inter-arrival times of the jobs are determined by the mean job processing time $\overline{p}$, the number of machines m in the system and a desired machine utilization rate U. If $\overline{p}$ is given, the mean inter-arrival time λ of the jobs can be calculated by $\lambda = \overline{p}/(mU)$. The inter-arrival times are then set using an exponential distribution with mean λ.

For this scenario, $\overline{p}$ was calculated by first calculating the mean processing times of the jobs in each machine ($\overline{p_i}$). We have then $\overline{p_1} = \overline{p_2} = \overline{p_4} = (12 + 28)/2 = 20$, and $\overline{p_3} = (5 + 50)/2 = 27.5$. Since there are two machine sequences a job can follow, (1,2,4) and (1,3,4), and the probability of a job to follow either one is 0.5, we derive that the mean processing time for the first route is $\overline{p_1} + \overline{p_2} + \overline{p_4} = 60$, and is $\overline{p_1} + \overline{p_3} + \overline{p_4} = 67.5$ for the second, ans so $\overline{p} = (60 + 67.5)/2 = 63.75$. Given the value of $\overline{p}$, we can calculate λ for each desired value of U.

The values of U used in our simulations were 0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85 and 0.95. These values represent from simpler, "lighter" working environments to more complex, "heavily loaded" ones.

4.2 Optimizing Criteria

The chosen optimizing criteria is the percentage of late jobs, which will be represented by the symbol $\overline{U}_{\%}$. We start by computing the number of late jobs in the system. For a job j, a function U_j is defined in [9] as

$$U_j = \begin{cases} 1 & \text{if } C_j > d_j \\ 0 & \text{otherwise} \end{cases}$$

and the number of late jobs, n_{t}, is defined as

$$n_{\mathrm{t}} = \sum_{j=1}^{n} U_j \ \ .$$

It is now easy to derive that the percentage of late jobs is calculated as

$$\overline{U}_{\%} = \frac{n_{\mathrm{t}}}{n} \ \ .$$

4.3 Simuluation Methodology

In the simulation, three different guessers were used, i.e., three different error factors for the guesser were considered: 10%, 20% and 30%, or mathematically, $f_{\mathrm{p}} \in \{0.1, 0.2, 0.3\}$. The GA with the 10%-guesser is called GA-10; the ones with the 20% and 30% guesser were named analogously, that is, GA-20 and GA-30. For the system's simulation the following methodology was used:

- Each simulation case will have 100 jobs.
- The jobs' processing times, due dates, release dates and routes were set as described in sections 4 and 4.1. The allowance factor is fixed with a value of 2 for every simulation case.
- Each simulation case was processed using all scheduling techniques mentioned so far.
- 50 different cases were simulated in total, for each utilization rate.

The average values for $\overline{U}_{\%}$ are calculated based on these 50 simulation runs for each utilization scenario.

4.4 GA Parametrization

The parametrization of the GA followed some well know values, as described in [1]. The crossover rate is 0.6 and the mutation probability is 0.1; also, the algorithm is run for 200 generations before it stops, and the population size is 50 individuals.

5 Results

The results concerning the simulations are summarized in tables 2 and 3. Table 2 shows the mean values achieved by each scheduling technique when trying to minimize the percentage of late jobs in the system. Table 3 combines data from a *pairwise t-test* with data from table 2. The pairwise t-test analysis was used to compare the best rule of each utilization scenario to the other techniques in order to show whether those can be considered statistically not different from the first with 95% confidence. To present this statistical analysis, table 3 has three values: *same*, *no* and *yes*. *Same* means that the results achieved by the two compared techniques are statistically not different; *no* indicates that the compared technique is worse than the best rule, and *yes* means that the compared technique is better than the best rule.

Util.	FIFO	EDD-ALL	SIRO	GA-10	GA-20	GA-30
0,15	0	0	0,04	0	0	0,02
0,25	0,08	0	0,14	0,08	0,16	0,14
0,35	0,68	0,18	0,96	0,56	0,54	0,52
0,45	1,78	0,54	2,46	1,2	1,14	1,42
0,55	8,58	4,64	7,46	4,6	4,64	4,8
0,65	22,48	15,9	18,94	12,2	12,66	12,76
0,75	39,6	30,86	28,78	20,8	21,18	22,9
0,85	59,88	55,28	45,5	34,56	35,98	37,04
0,95	75,54	70,6	58,28	46,36	46,4	47,88

Table 2: Mean percentage of late jobs for the scheduling techniques

Util.	Best Rule	FIFO	EDD-ALL	SIRO	GA-10	GA-20	GA-30
0.15	EDD	same	-	same	same	same	same
0.25	EDD	no	-	no	no	no	no
0.35	EDD	no	-	no	no	no	no
0.45	EDD	no	-	no	no	no	no
0.55	EDD	no	-	no	same	same	same
0.65	EDD	no	-	no	yes	yes	yes
0.75	SIRO	no	same	-	yes	yes	yes
0.85	SIRO	no	no	-	yes	yes	yes
0.95	SIRO	no	no	-	yes	yes	yes

Table 3: Better than the best rule with 95% confidence

From table 2 it can be seen that the guess and solve approach achieves better results for scenarios with utilization rates ranging from 0.65 to 0.95. It also noticeable that the EDD is the best rule until the value of U reaches 0.65, from where the SIRO rule takes over the first place. Detailed study of the simulation revealed that as the value of U raises, the jobs' due dates become tighter, and this forces every newly arrived job j to spend much more

time waiting in the machines' queues. These jobs get no more allowance since the allowance factor is fixed for all simulations; this situation leads to a point where the EDD rule is not capable of choosing a job which is not late, since these jobs are already too late. When SIRO is used, the job to be executed is chosen at random, so a certainly late job within the EDD context now has a chance to be chosen before other jobs and leave the system before its due date is reached.

Table 3 shows that the guess and solve approach remains worse than the rules for values of U ranging from 0.15 to 0.45. When a utilization rate of 0.55 is reached, the guess and solve catches up with the best rule's performance, and when U reaches 0.65 it becomes better than the best rule, keeping this first place until $U = 0.95$. Since systems with reasonable workloads are expected to have their utilization rates above 0.65, the guess and solve shows itself as a good technique for the common medium to heavily loaded systems of today's workplaces.

Figures 2 and 3 show the evolution of the scheduling techniques for all utilization scenarios. It can be seen that even being considered statistically worse than the best rule in lower utilization scenarios, the guess and solve is always near its results, confirming itself as a good technique even when $U < 0.65$.

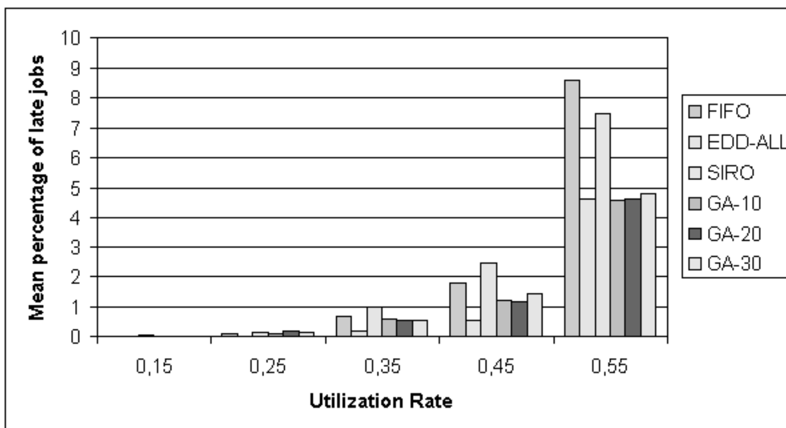

Figure 2: Mean percentage of late jobs for utilization rates from 0.15 to 0.55

6 Conclusions and Future Work

The job shop is a very important scheduling problem, but research on it mainly concentrates in deterministic instances. This report presents a convenient way to use genetic algorithms with processing time uncertainties in a JSSP and also a way to adapt the dynamic-into-static decomposition approach to these uncertainties.

The numerical results from the simulations show that the EDD rule prevails over the other rules until $U = 0.65$, when the SIRO rule becomes the best rule. Also, the guess and solve approach together with the adjustment to the decomposition technique follows

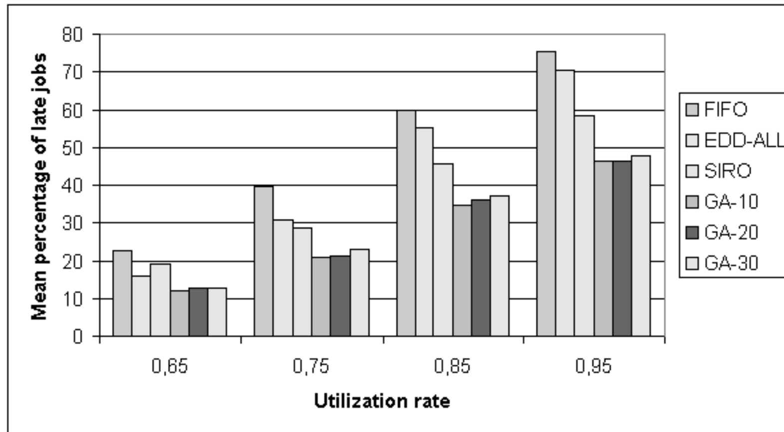

Figure 3: Mean percentage of late jobs for utilization rates from 0.65 to 0.95

the best rules' behavior closely, surpassing their performance when U reaches 0.65, showing that this combination is a fairly adequate approach to solve the JSSP.

Future work would involve analyzing the guess and solve in various different scheduling scenarios, and also with different error factors. Guesses that get better with gaining experience on the system's past cases should be modeled as a increasing precision guesser and should also be subject to research. Uncertainties regarding the routes the jobs follow through the system, which are a quite common characteristic of some real world situations, should also be considered as research subjects.

References

[1] Christian Bierwirth, Herbert Kopfer, Dirk C. Mattfeld, and Ivo Rixen. Genetic algorithm based scheduling in a dynamic manufacturing environment. In *Proc. of 1995 IEEE Conf. on Evolutionary Computation*, Piscataway, NJ, 1995. IEEE Press.

[2] Christian Bierwirth and Dirk C. Mattfeld. Minimizing job tardiness: Priority rules vs. adaptative scheduling. In *Adaptative Computing in Design and Manufacture*, London, 1998. Springer-Verlag.

[3] Christian Bierwirth and Dirk C. Mattfeld. Production scheduling and rescheduling with genetic algorithms. *Evolutionary Computation*, 7:1–17, 1999.

[4] Hsiao-Lan Fang, Peter Ross, and Dave Corne. A promising genetic algorithm approach to job-shop scheduling, rescheduling, and open-shop scheduling problems. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 375–382. Morgan Kaufmann, 1993.

[5] Paulo M. França, Alexandre Mendes, and Pablo Moscato. A memetic algorithm for the total tardiness single machine scheduling problem. *European Journal of Operational Research*, 132:224–242, 2001.

[6] Anant Singh Jain and Sheik Meeran. 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, 1998.

[7] Manuel Laguna. Business process modeling, simulation and design. (forthcoming).

[8] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[9] Michael Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[10] P. Pongcharoen, C. Hicks, P. M. Braiden, and D. J. Stewardson. Determining optimum genetic algorithm parameters for scheduling the manufacturing and assembly of complex products. *International Journal of Production Economics*, 78:311–322, 2002.

[11] N. Raman and F. Brian Talbot. The job shop tardiness problem: A decomposition approach. *European Journal of Operational Research*, 69(2):187–199, 1993.