Simulating Stochastic Scheduling Policies on Unrelated Machines

Tariq Bontekoe

June 26, 2017

Abstract

We investigate the competitive performance bounds of a purely combinatorial, online algorithm for stochastic and non-preemptive scheduling on unrelated machines. Several simulation experiments with different instances are executed and the performance is investigated from a computational perspective. The performance of this online algorithm is compared to an offline LP relaxation based algorithm and the WSEPT rule, both on average and in worst case. Finally, the random LP-relaxation based policy is derandomized and the effects on the performance of this algorithm are investigated.

1 Introduction

Scheduling. Scheduling n jobs on m parallel machines is a fundamental problem in combinatorial optimization. The problem to minimize the total weighted completion time $\Sigma_j w_j C_j$ on unrelated machines, denoted by $\mathrm{R} \mid (r_j) \mid \Sigma_j w_j C_j$ in the three-field notation of Graham et al. [1], is one of the classical problems in *deterministic scheduling*. In this problem, each job j has a weight w_j, and a processing time on machine i, which is defined by p_{ij}. A job j also might have a release date, which is given by r_j, before which job j cannot be scheduled on any machine. The objective of this problem is to minimize the total weighted completion time $\Sigma_j w_j C_j$, where C_j denotes the completion time of job j. This problem turned out to be MaxSNP-hard [2], which implies that no polynomial-time approximation algorithm for a solution to this problem exists.

In a lot of cases, the processing time p_{ij} of job is unknown and uncertain. This is where deterministic scheduling becomes *stochastic scheduling*. Unfortunately, this uncertainty makes the scheduling of jobs significantly harder than in the deterministic case. We would then still want to be able to come up with a schedule which minimizes the weighted completion time. So, we would like to find a non-anticipatory scheduling policy which schedules n non-preemptive jobs with uncertain processing times p_{ij} on m machine. Here, *non-anticipatory* means that for a decision at time t the scheduling policy can only use information based on the observations up to time t. The fact that jobs are *non-*

preemptive means that once a job is started, it cannot be interrupted and is being processed until it is finished.

This problem R | (r_j) | $\Sigma_j w_j C_j$ with unknown processing times p_{ij} has both an offline and an online version. This problem was for the first time investigated by Skutella et al. [3] and resulted in an algorithm with a worst case performance guarantee of $\frac{3}{2} + \frac{\Delta}{2}$ without release dates and a guarantee of $2+\Delta$ when release dates are present. In the *offline* version the entire set of jobs is presented to the algorithm at the start. In the *online* version the jobs are presented to the scheduling algorithm one by one and the magnitude of the complete set of jobs is unknown to the scheduler.

In this paper, we consider the online stochastic version of the unrelated machine problem, denoted by R | (r_j) | $\Sigma_j w_j C_j$. There are several approaches to tackle this problem. It is known however, that when n jobs appear online, are non-preemptive, and nothing else is known about these jobs, no algorithm can be better than n-competitive. The fact that an online algorithm ALG is α-competitive, means that for any given instance I, cost function $\mathcal{C}$ and constant c: $\mathcal{C}(ALG(I)) \leq \alpha \cdot \mathcal{C}(OPT(I)) + c$, where OPT is the optimal offline algorithm for this instance I. In the case of stochastic scheduling, the cost function $\mathcal{C}$ equals the expected weighted completion time $\mathbb{E}[\sum_{j\in J} w_j C_j]$.

One solution to this problem is to assume that jobs may be pre-empted. Many papers have been written on this type of model. Another solution is the one we use in this paper: the jobs remain non-preemptive, but we know the probability distribution of p_{ij}. This is called the stochastic scheduling model with non-preemptive jobs. Since, in this paper we assume that we know the probability distribution of the processing time of job, we will from now on use P_{ij} to denote the random variable belonging to the processing time, whereas p_{ij} will be used to represent the realization of P_{ij}.

Contribution. Throughout this paper, we take a look at a new, very recent algorithm from Gupta et al. [4], which is the first online, $O(\Delta)$-competitive, combinatorial algorithm for stochastic scheduling on unrelated machines. Here, Δ is the square of the coefficient of variation $\mathbb{CV}^2[P_{ij}] = \mathbb{V}\text{ar}[P_{ij}]/(\mathbb{E}[P_{ij}])^2$. They derive an algorithm which is $(8+4\Delta)$-competitive in the case without release dates, this is called the *online-list model*. When release dates are present, the *online-time model*, a $(144+72\Delta)$-competitive bound is derived. These bounds may seem quite large, but it should be taken into account that this algorithm is purely combinatorial and hence, is computationally attractive. This is contrary to the algorithm for the online case from Skutella et al. [3], which involves a computationally cumbersome time-indexed linear programming relaxation. Finding the optimal solution for these kinds of LP relaxations is generally hard to achieve in practice.

In this paper, the tightness of the bounds of the online-list model from Gupta et al. [4] is investigated. By this, we mean that we will take a look at the optimality gap, the gap between the weighted completion time of the optimal offline algorithm and the weighted completion time of this online, combinatorial algorithm. We will look at this optimality gap from a computational perspective.

The combinatorial algorithm [4] will be compared to offline algorithms from [3] and the optimal value of the LP relaxation from the same paper. We will look at the optimality gap of the combinatorial algorithm on 'regular', as well as (possibly) 'problematic' instances. Throughout this paper, several experiments with these instances are performed and computational bounds on the optimality gaps for these instances will be shown. By performing these computational experiments, we analyze whether the provided bound of $(8+4\Delta)$ might be too loose and whether the algorithm performs better than these bounds imply or indeed behaves as badly.

Finally, we will also look at the randomized algorithm provided for the offline case [3]. The performance bound for this is shown to be $\frac{3}{2}+\frac{\Delta}{2}$. Throughout the experiments we will take a look at the tightness of the performance bound in practice. Also, we will look at the effect on the practical performance of a derandomized version of this algorithm. This derandomization will be performed using the well-known method of conditional probabilities [5]. We will show that is algorithm works in polynomial time, has equal or better performance than the random version and, we will simulate the performance of this derandomized algorithm.

2 Notation and Preliminaries

Throughout this paper every instance of a stochastic scheduling model will be defined as follows. We are given a set of jobs J of cardinality n with job weights $w_j \in \mathbb{R}_{>0}, j \in J$, and a set of unrelated parallel machines M of cardinality m. Moreover, we are given a random variable P_{ij} which represents the distribution of the processing time of job $j \in J$ on machine $i \in M$. Each job j must be executed non-preemptively on one of the machines $i \in M$ and each machine $i \in M$ can process at most job at the same time. If a job j is being processed on machine i its processing time is given by P_{ij} however the actual realisation p_{ij} of this processing time P_{ij} is only known after the job has finished. For each instance we will construct a non-anticipatory scheduling policy which minimizes the expected total weighted completion time $\mathbb{E}[\sum_j w_j C_j]$, where C_j denotes the completion time of job j. Here, the expectancy is taken over C_j which is dependent of the random variables P_{ij}. We take the expected value because we want too know how the policy performs in expectation and not on some particular realization of P_{ij}'s.

The fact that the jobs are non-preemptive means that once a job is started it cannot be interrupted until it has been processed. The fact the the scheduling policy has to be non-anticipatory means that at time t the policy can only use information that has been observed upto time t. It is allowed that certain jobs $j \in J$ cannot be processed on certain machines $i \in M$, in which case $\mathbb{E}[P_{ij}] = \infty$.

Furthermore we will make use of the online-list model, meaning that the existence of job j is unknown before it is shown to the machine, and upon arrival, only the random variables P_{ij} are known. A non-anticipatory policy will, at any time t, decide whether a job j should be assigned to a machine i based on the information that has been observed in the past upto time t. Without loss of generality we also assume that no pair of job j and machine i exists such that $\mathbb{E}[P_{ij}] = 0$, as then job j can always be scheduled at machine i at no cost.

3 Stochastic Scheduling Policies for Unrelated Machines

In this section we will take a look at two algorithms for scheduling policies on unrelated machines. Firstly, an offline time-indexed LP relaxation with accompanying scheduling algorithm as described in [3] will be stated. Secondly, the online greedy combinatorial algorithm from [4] will be stated. The latter algorithm is the one we are interested in and the former one will provide us with a means of comparison. The time-indexed LP relaxation will be used as a comparative in two ways.

Firstly, we use the optimal solution of the LP relaxation. The fact that this is an LP relaxation implies that the optimal value hereof will be less than or equal to the optimal value of any non-anticipatory schedule. This makes that we can use this optimum to provide a lower bound for the expected weighted completion time of an optimal policy. The second way in which we can use the LP relaxation as a comparative, is by using the accompanying scheduling algorithm denoted by $\textsc{Assign}(X)$. This policy is constructed by using the solution of the LP relaxation and is currently the best known offline scheduling algorithm for the stochastic scheduling problem on unrelated machines. This means that it can serve as an adversary for the online algorithm as described in [4] and provide an upper bound for the expected weighted completion time of an optimal policy. In our experiments we will use both this lowerand upper bound for the optimal expected weighted completion time to analyze the performance of the online, purely combinatorial algorithm from [4].

3.1 Offline Time-indexed LP relaxation

The LP relaxation of the stochastic scheduling model for unrelated machines assumes that the random variables $P_{ij}, i \in M, j \in J$ take positive integral values only. By [3, Lemma 1] this costs at most a factor $1+\epsilon$ for any fixed $\epsilon > 0$ in the objective function value. A certain factor ϵ can be achieved by scaling the time appropriately. If one, for example, multiplies all processing times by a factor 10, the amount of time variables in the LP gets 10 times as big. But, actually this is similar as taking steps of $\frac{1}{10}$ in time instead of steps with size 1, which is normal. So, by scaling the time the LP relaxation becomes more accurate. In this paper we will from now on assume that P_{ij} takes positive integral values. If P_{ij} does not take integral values we will use a discrete approximation of P_{ij}.

This implies that, without loss of generality, jobs may only be started at an integral point in time $t \in \mathbb{Z}_{\geq 0}$, at the cost of a factor ϵ. The other information about the processing time P_{ij} that is needed to formulate the LP relaxation are the expected processing times P_{ij} and the values

$$q_{ijr} := \mathbb{P}[P_{ij} \geq r+1] \qquad \text{for} i \in M, j \in J, r \in \mathbb{Z}_{\geq 0}$$

Below follows the LP relaxation of the given problem for unrelated machines as derived in [3, §3]. Let x_{ijt} be the probability that job $j \in J$ is started on machine $i \in M$ at time $t \in \mathbb{Z}_{\geq 0}$. This means that this decision may depend on the actual processing times of jobs started beforehand. It is however independent of decisions that happen at time $t^* \geq t$. These probabilities x_{ijt} are the variables of the LP relaxation. The LP relaxation, which results in an optimal solution x_{ijt} corresponding to a policy Π (this policy is a solution to the LP relaxation problem, it might not be a solution to the original scheduling problem) where the expected weighted completion time is equal to the optimal value of the LP solution x, is:

$$\begin{aligned}
\min \quad & \sum_{j\in J} w_j C_j^{LP} & & \qquad (1)\\
\text{s.t.} \quad & \sum_{i\in M}\sum_{t\in\mathbb{Z}_{\geq 0}} x_{ijt} = 1 & \forall \ j \in J, & \qquad (2)\\
& \sum_{j\in J}\sum_{t=0}^{s} x_{ijt} q_{ijs-t} \leq 1 & \forall \ i \in M, s \in \mathbb{Z}_{\geq 0}, & \qquad (3)\\
& C_j^{\text{LP}} = \sum_{i\in M}\sum_{t\in\mathbb{Z}_{\geq 0}} x_{ijt}(t + \mathbb{E}[P_{ij}]) & \forall \ j \in J, & \qquad (4)\\
& x_{ijt} \geq 0 & \forall \ j \in J, i \in M, t \in \mathbb{Z}_{\geq 0}. &
\end{aligned}$$

In the above defined LP relaxation equations (1) and (4) together define the objective. Since the LP variables C_{LP} are uniquely defined by x_{ijt} this might also be placed in the objective function (1), which represents the expected weighted completion time.

The constraints are given by equations (2) and (3). The former ensures that every job is processed completely. The latter constraint ensures that that the expected number of jobs processed by machine i during timespan $[s, s+1]$ is no more than 1.

It should be remarked that this LP has an infinite amount of variables x_{ijt} due to the fact that for any time $t \geq 0$ the probability that a job can be started may be positive. Fortunately, there exists a bound T such that an optimal solution with $x_{ijt} = 0, t > T$ exists. This bound [3, Lemma 3] is given by:

$$T := 2n \max_{j\in J, i\in M} \mathbb{E}[P_{ij}] + 2\sum_{j\in J} w_j \cdot \max_{i\in M} \sum_{j\in J} \mathbb{E}[P_{ij}] \qquad (5)$$

By applying this bound, the amount of variables becomes finite and it is now possible to practically obtain an LP solution. In order to transform the LP solution into a scheduling policy, we use the policy $\textsc{Assign}(X)$ [3, §4]. Here X is defined by using the feasible LP solution X, by defining $X_{ij} := \sum_{t\in\mathbb{Z}_{\geq 0}} x_{ijt}$ for $i \in M$, $j \in J$. The scheduling policy $\textsc{Assign}(X)$ now assigns each job $j \in J$ independently at random to one machine $i \in M$ with probability X_{ij}. After all jobs have been assigned, the jobs get sorted in the right order. On each machine $i \in M$ the jobs assigned to i are scheduled according to the WSEPT (weighted shorted expected processing time) rule. So, in non-increasing order of $w_j/\mathbb{E}[P_{ij}]$.

It is known that the expected value of the schedule constructed by $\textsc{Assign}(X)$ is at most a factor $\frac{3}{2} + \frac{\Delta}{2}$ times the value of the underlying LP solution x [3].

3.2 Online Greedy Algorithm

For the online Greedy Algorithm [4] it is w.l.o.g. assumed that the jobs are presented in the order $1, 2, \ldots, |J|$. On any machine i, let $H(j,i)$ and $L(j,i)$ denote the sets of all jobs that have higher respectively lower priority than job j according to their order in non-increasing order of ratios $w_j/\mathbb{E}[P_{ij}]$, ties broken by index:

$$\begin{aligned} H(j,i) &:= \{k \in J \mid \frac{w_k}{\mathbb{E}[P_{ik}]} > \frac{w_j}{\mathbb{E}[P_{ij}]}\} \cup \{k \in J \mid k \leq j,\ \frac{w_k}{\mathbb{E}[P_{ik}]} = \frac{w_j}{\mathbb{E}[P_{ij}]}\} \\ L(j,i) &:= J \setminus H(j,i) \end{aligned}$$

Also, let $k \to i$ denote the fact that job k has been assigned to machine i. With these definitions the Greedy Algorithm works as follows: Whenever a new job $j \in J$ is presented to the Greedy Algorithm, the algorithm calculates the expected increase $\mathrm{EI}(j \to i)$ for all machines $i \in M$ by means of equation (6).

$$\mathrm{EI}(j \to i) := w_j \left(\mathbb{E}[P_{ij}] + \sum_{k \to i, k<j, k \in H(j,i)} \mathbb{E}[P_{ik}] \right) + \mathbb{E}[P_{ij}] \sum_{k \to i, k<j, k \in L(j,i)} w_k \quad (6)$$

The Greedy Algorithm now assigns job j to the machine where the expected increase is minimal, i.e. j is assigned to machine $i(j) := \mathrm{argmin}_{i \in M}\{\mathrm{EI}(j \to i)\}$, ties broken arbitrarily. Once all jobs have been assigned to a machine, they are sequenced according to the WSEPT rule, so in non-increasing order of $w_j/\mathbb{E}[P_{ij}]$. In [4, Theorem 1] it is proven that this algorithm is $(8 + 4\Delta)$-competitive for this stochastic scheduling problem. The competitive ratio holds with respect to the worst-case sequence of online jobs and processing time distributions. It does however not hold for the worst realization of processing times P_{ij}, but only for the expected processing times.

4 Analysis on Identical Machines

In this section we take a look at the performance of both the greedy algorithm and the LP relaxation on the stochastic simulation problem with identical parallel machines. This problem is in principle easier than the problem with unrelated machines, but a good comparative algorithm already exists for this problem: the offline WSEPT rule (schedule jobs with highest ratio of weight to expected processing time first). The performance bound for the WSEPT-rule is $1+\frac{(m-1)(\Delta+1)}{2m}$ for arbitrary distributions of P_{ij} and $2-\frac{1}{m}$ if $\mathbb{CV}[P_{ij}] \leq 1$ [6]. This guarantee is quite a lot better than the guarantee of the online Greedy Algorithm, so it is interesting to make this comparison.

4.1 Experiment

In order to evaluate the practical performance of the Greedy algorithm, we compare its expected performance to the solution of the LP relaxation (optimal value & policy). Besides that, we are also interested in the expected performance of the WSEPT rule, even though that rule is not an online algorithm. This will be considered as an additional benchmark. All algorithms were programmed in Python and this code can be found in Appendix B. Firstly, some comparisons are made using general instances in order to review the general performance of the Greedy Algorithm.

These general instances have different amounts of machines and jobs, with exponentially distributed processing times P_{ij}. This instance type is defined as follows:

Instance 1.

$$\begin{aligned} &P_{ij} \sim \text{Exp}(k_i) && k_i \in \mathbb{Z}, 1 \leq k_i \leq 10, \forall j \in J \\ &P_{ij} = P_{mj} && \forall i, m \in M, \forall j \in J \\ &w_j = 1 && \forall j \in J \end{aligned}$$

The expected value of these processing times is allowed to take integral values from one upto and including 10, i.e. $0 \leq \mathbb{E}[P_{ij}] \leq 10, \mathbb{E}[P_{ij}] \in \mathbb{Z}$. Notice that in the case of exponentially distributed processing times $\Delta = 1$. Since all machines are identical: $P_{i_1 j} = P_{i_2 j} \;\; \forall i_i, i_2 \in M$. The amount of jobs reaches from 1 to 9 and the amount of machines from 2 to 4. Per combination of amount of jobs and machines 100 instances, with randomly drawn values of $\mathbb{E}[P_{ij}]$ in the given domain, were used as input. Per instance the optimal values of the WSEPT rule, Greedy Algorithm, LP and LP ASSIGN(X)-algorithm were compared. The calculation of these values will be explained in the next paragraph

Also simulations with bigger amounts of jobs and machines were executed. Since the calculations on the LP are computationally very involved simulations of bigger versions of Instance 1 are done without comparing to the optimal LP value or the ASSIGN(X) policy. The amount of machines in this case ranges from 1 to 100 and the amount of jobs from 20 to 500.

After the first instances, we will take a look at a known problem instance for the WSEPT rule, namely the stochastic Kawaguchi and Kyan instance [7]. It is interesting to see if this instance is also problematic for the Greedy Algorithm. Next to that, it is interesting to see if there exist problematic instances which come close to the $8+4\Delta$ performance bound. The instance is defined as follows, where $h \in \mathbb{Z}$ and $k \in \mathbb{Z}$ is divisible by $\lfloor(1+\sqrt{2})h\rfloor$:

Instance 2 (Stochastic Kawaguchi Kyan).

$$\begin{aligned}
|M| &= h + \lfloor(1+\sqrt{2})h\rfloor \\
|J| &= mk + h \\
P_{ij} &\sim \mathrm{Exp}(1/(1+\sqrt{2})) && \text{for } 1 \le j \le mk,\ \forall i \in M \\
w_j &= 1+\sqrt{2} && \text{for } 1 \le j \le mk,\ \forall j \in J \\
P_{ij} &\sim \mathrm{Exp}(k) && \text{for } mk+1 \le j \le mk+h,\ \forall i \in M \\
w_j &= 1/k && \text{for } mk+1 \le j \le mk+h,\ \forall j \in J
\end{aligned}$$

Notice that $w_j/\mathbb{E}[P_{ij}] = 1$ for all jobs. This implies that every list schedule is a WSEPT schedule, however the order in which the jobs are scheduled does influence the expected weighted completion time. The schedule has optimal performance when all long jobs are scheduled fist, and worst-case performance if the long jobs are scheduled last. This instance gives a worst case performance bound of approximately 1.229, which is worse than the worst known deterministic instance which has a tight performance bound of approximately 1.207 [8]. This instance is simulated with a few different amounts of machines and jobs. For each simulation a comparison between the optimal list schedule for this instance and respectively the worst case schedule for the WSEPT rule or the Greedy Algorithm is made.

Also another type of possibly problematic instances are simulated for the Greedy Algorithm with respect to the WSEPT algorithm. These instances consist of jobs having a so called 2-point distribution. In this case all instances had the following processing time distributions P_{ij}, for different probabilities p:

Instance 3.

$$\begin{aligned}
P_{ij} &= \begin{cases} 1, & \text{with probability } p \\ 10^{10}+1, & \text{with probability } p \end{cases} && 0.5 \le p < 1, \forall j \in J \\
w_j &= 1 && \forall j \in J
\end{aligned}$$

Each instance will have 10 machines, a different value for Δ, and a different amount of jobs ranging from 20 to 100. The factor Δ can be influenced by changing p, since for this instance $\Delta \approx \frac{p}{1-p}$. Here, p must be chosen in such a way that $(1-p)\cdot 10^{10}$ is a lot bigger than 1, in the experiment $0.5 \le p \le 64/65$, so this condition holds throughout the experiment. This fact is derived as follows:

$$\begin{aligned}
\mathbb{E}[P_{ij}] &= 1 + (1-p)\cdot 10^{10} \\
\mathbb{V}\mathrm{ar}[P_{ij}] &= p((1-p)\cdot 10^{10})^2 + (1-p)(1+p\cdot 10^{10})^2 \\
\Delta &= \frac{\mathbb{V}\mathrm{ar}[P_{ij}]}{(\mathbb{E}[P_{ij}])^2} \\
&= \frac{p((1-p)\cdot 10^{10})^2 + (1-p)(1+p\cdot 10^{10})^2}{(1+(1-p)\cdot 10^{10})^2} \\
&\approx \frac{p((1-p)\cdot 10^{10})^2 + (1-p)(p\cdot 10^{10})^2}{((1-p)\cdot 10^{10})^2} \qquad \text{(provided } 1 \ll (1-p)\cdot 10^{10}\text{)} \\
&= \frac{p(1-p)+p^2}{1-p} \\
&= \frac{p}{1-p}
\end{aligned}$$

Instance 3 has jobs with high variance and it is interesting to see how the multiplicative optimality gap between the WSEPT rule and the Greedy Algorithm differs with the amount of jobs and Δ. Especially, since the proven performance bound for the Greedy Algorithm contains a factor 4Δ.

Also, a few other instance types were considered but eventually not included in this paper. We did not include deterministic instances, or instances with a very low variance, since less uncertainty gives less interesting results. Neither did we include instances which, just like Instance 3 have a high coefficient of variation, since this instance already provides us with enough information regarding Δ. Lastly, we did not try to perturb and/or combine instances mentioned above, since we are mainly interested in the performance of the Greedy Algorithm on unrelated machines. A few test simulations were performed, but no new or additive information was gained.

4.2 Constructing the Experiment

As mentioned in the previous paragraph, simulations were run by means of code written in Python (see Appendix B). The way this code implements the simulation of the different algorithms shall be explained roughly below, as to sketch the execution of the experiments.

A simulation of a particular instance starts with the generation of the instance according to the mentioned specifications. This means that the machines, jobs, job weights and job distributions are instantiated. The real processing times are still unknown. When the instance is instantiated, it is passed to (a selection of) the algorithms.

The **WSEPT algorithm** obtains the instance and schedules the job according to the WSEPT rule (weighted shorted expected processing time first) [9, §10.1]. The WSEPT rule works as defined in Algorithm 1.

We see there that jobs get assigned to a machine one by one.Once they are assigned a realization p_{ij} of the processing time P_{ij} is drawn from the given distribution. This process continues until all jobs have been assigned. Afterwards the weighted completion times get summed up, which results in the total weighted completion time. Since the realized processing times p_{ij} are drawn randomly from a given distribution, the same instance is processed by the WSEPT rule 10,000 times with different, randomly drawn, realizations of the processing times. By taking the average of the weighted completion time of each run, we obtain the expected weighted completion time $\mathbb{E}[\sum_{j\in J} w_j C_j]$ of the given instance.

```
Data: Jobs: $J$; Machines: $M$
Result: Policy: $\Pi$; Weighted Completion Time: $value$
initialize $t := 0$;
while there are unscheduled jobs do
    $job :=$ argmax$_{j\in J}\{\frac{w_j}{\mathbb{E}[P_{ij}]}\}$;
    if a machine is idle at time t then
        schedule $job$ at time $t$ at an idle machine;
        draw a realization $p_{ij}$ of the processing time;
    else
        change $t$ to next time a machine falls idle;
        $value = value + w_j \cdot t$, where $j$ is the job that finished at time $t$;
    end
end
while jobs are being processed do
    change $t$ to next time a machine falls idle;
    $value = value + w_j \cdot t$, where $j$ is the job that finished at time $t$;
end
```

Algorithm 1: Calculating the weighted completion time of an instance for the WSEPT rule

The **Greedy Algorithm** is deterministic in the sense that for every instance it will return one schedule, independent of the realizations p_{ij} of the processing times P_{ij}. Because of this, the expected weighted completion time $\mathbb{E}[\sum_{j\in J} w_j C_j]$ can be calculated by looking at the weighted completion time of the same instance, when using the expected processing time $\mathbb{E}[P_{ij}]$ instead of the realizations p_{ij}. This agrees with [4, Lemma 3]. So during the simulation, firstly the jobs are presented in their input order to the algorithm, which schedules the jobs to the machines. Once all jobs are scheduled the expected completion time of each machine is calculated and these are added together. This also results in the correct value for the expected weighted completion time $\mathbb{E}[\sum_{j\in J} w_j C_j]$. This entire process is depicted by Algorithm 2.

```
Data: Jobs: $J$; Machines: $M$
Result: Policy: $\Pi$; Expected weighted completion time: $value$
for $j \in J$ do
    $j \to \mathrm{argmin}_{i \in M}\{\mathrm{EI}(j \to i)\}$;
end
for $i \in M$ do
    Order all jobs $j$ assigned to $i$ in non-increasing order of $\frac{w_j}{\mathbb{E}[P_{ij}]}$;
end
$value := 0$;
for $i \in M$ do
    $t := 0$;
    for $j$ assigned to $i$ (in order defined previously) do
        $t = t + \mathbb{E}[P_{ij}]$;
        $value = value + w_j \cdot t$;
    end
end
```

Algorithm 2: Calculating the expected weighted completion time of the Greedy Algorithm

The **Linear Programming Relaxation** obtains the instance and starts with constructing the time-indexed LP relaxation defined by equations (1), (2), (3) and (4). This LP is than solved using Gurobi and its Python API. This will return an optimal value, which is smaller than or equal to the expected weighted completion time of an optimal policy for the given instance.

The optimal value of the LP relaxation is used as a lower bound for the actual optimal solution of the given instance. The optimal LP relaxation solution X is also used in the LP-based algorithm $\textsc{Assign}(X)$. Recall that X is defined by, $X_{ij} := \sum_{t \in \mathbb{Z}_{\geq 0}} x_{ijt}$ for $i \in M$, $j \in J$. Since this optimal solution only uses probabilistic information concerning the processing times P_{ij}, it is independent of the realizations p_{ij} of the processing times. Because of this fact, also here we will use the expected processing time $\mathbb{E}[P_{ij}]$ instead of the realizations p_{ij} thereof. This algorithm must still be run multiple times (10,000) since the policy $\textsc{Assign}(X)$ contains a random element (it assigns jobs to machines according to the probability distribution X). The average value of all runs is calculated and this results in the expected weighted completion time $\mathbb{E}[\sum_{j \in J} w_j C_j]$ of the policy $\textsc{Assign}(X)$. The process of one run is depicted in Algorithm 3.

```
Data: LP solution $X$; Machines: $M$; Jobs: $J$
Result: Policy: $\Pi$; Policy value: $value$
for $j \in J$ do
    $j \rightarrow i$ with probability $X_{ij}$;
end
for $i \in M$ do
    Order all jobs $j$ assigned to $i$ in non-increasing order of $\frac{w_j}{\mathbb{E}[P_{ij}]}$;
end
$value := 0$;
for $i \in M$ do
    $t := 0$;
    for $j$ assigned to $i$ (in order defined previously) do
        $t = t + \mathbb{E}[P_{ij}]$;
        $value = value + w_j \cdot t$;
    end
end
```
Algorithm 3: Calculating the weighted completion time of ASSIGN(X)

4.3 Results & Analysis

In the previous paragraph, the experiment for the identical machine case was described. Firstly, we will take a look at Instance 1. These are instances with an amount of jobs between 1 and 9 and an amount of machines reaching from 2 to 4, where the jobs are exponentially distributed with weights of 1. Since it is hard to know what the actual optimal value is, we will take a look at the multiplicative factor between the optimal solution according to the LP relaxation and the expected weighted completion time of each algorithm. The value of the optimal LP solution might be unreachable in reality, but it does give a good look into how far each algorithm is maximally away from the optimum. Besides that, the LP solution provides a good benchmark in order to make a comparison between all algorithms.

The results of the simulation can be found in Table 1 in the appendix. This table shows for each amount of machines and jobs, what the average respectively worst case factor of difference is between the LP solution and the used algorithm. The first thing that is noticeable is the fact that the WSEPT algorithm behaves best on average and in worst case for all the instances. The performance of the Greedy Algorithm is less with respect to the WSEPT rule, the difference factor is about 0.1 for the average case and the worst case. ASSIGN(X) clearly behaves clearly worse, the factor of this policy is approximately 0.2 worse than the WSEPT rule, in both the average and worst case.

Observation 1. *The Greedy Algorithm has a factor* ≈ 1.1 *better performance than* ASSIGN(X) *on 'average' instances for identical machines.*

So, as expected the WSEPT rule has the best performance guarantee on small average instances. This is to be expected since it makes use of the fact that jobs sometimes finish sooner than expected and does not schedule all jobs at time 0, which is what the Greedy and $\textsc{Assign}(X)$ algorithms do. However, it is also interesting that the relative difference in performance guarantee becomes smaller as the amount of jobs increases.

When looking at bigger amounts of jobs and machines we obtain Table 2 in the appendix. This shows with what multiplicative factor the value of the Greedy Algorithm is higher than that of the WSEPT rule. On average the factor of difference seems to lie around 1.15 and the worst case difference in the simulation was about 1.17. We see that this factor is not really that dependent on the amount of jobs or machines and that the difference remains quite small.

> **Observation 2.** *Both the Greedy Algorithm and* $\textsc{Assign}(X)$ *have notable worse performance on 'average' instances for identical machines than the WSEPT algorithm.*

After the experiment on normal cases, an experiment with a known worst-case instance for the WSEPT-rule was tested, the stochastic Kawaguchi Kyan instance (Instance 2). The results can be found in Table 3. This instance was run with different amounts of machines |M| and different amounts of jobs. Here we see that the worst case performance guarantee of the WSEPT rule is a little bit lower than that of the Greedy Algorithm. However the worst case performance guarantee of the WSEPT rule increases when the instance becomes bigger and as shown in [7] will be approximately 1.23 when enough machines and jobs are present. The Greedy Algorithm seems to have the worst performance when the amount of jobs is a bit lower (1.25) and slightly better when this amount increases (1.23). This can be explained by the fact that for a big amount of jobs it does not really matter that much that all jobs were scheduled at time $t = 0$. However, this difference is minor, and it seems that the worst case performance guarantee of the Greedy Algorithm is quite similar to that of WSEPT for this worst case instance.

> **Observation 3.** *The Greedy Algorithm has slightly worse performance on the difficult stochastic Kawaguchi Kyan instance than the WSEPT rule. When the amount of jobs and machines increases, they behave similarly and the Greedy Algorithm is a good alternative here.*

Actually the performance of the Greedy Algorithm seems to be overall very close to WSEPT, since both algorithms work similarly. WSEPT can however have a small advantage due to the fact that jobs are only scheduled when a machine falls idle. This gives the possibility to use the fact that jobs might end sooner than expected due to their stochastic processing times. The Greedy Algorithm does not have this possibility, but the jobs are scheduled in a similar way, so the worst case performance of this algorithm should not be a lot worse than that of the WSEPT rule.

Finally we look at Instance 3, with high variance. The multiplicative gap between the WSEPT rule and the Greedy Algorithm is investigated. The results of the experiment can be found in Table 4. Firstly, we observe that for each Δ their exists an amount of jobs which gives the biggest optimality gap. This is quite logical, since with a really high amount of jobs almost each machine will have a job with a very long processing time, and in that case it does not really matter whether all jobs are assigned at time 0 or only when a machine falls idle.

Observation 4. *The Greedy Algorithm performs a lot worse than the WSEPT rule, on instances with a high coefficient of variation and identical machines. The optimality gap between the two algorithms increases positively with* Δ.

We also see here that increasing Δ increases the gap notably. When we look closely, we see that the gap does not increase with a factor 8 when Δ is doubled. In the worst case, the gap increases at the same rate as Δ, but most of the time it increases way slower. This makes us realize that the factor 4Δ in the proven performance bound is too negative. The algorithm probably has a way lower performance bound of a constant plus 1 or 2 Δ.

Observation 5. *The optimality gap of the Greedy Algorithm most likely does not increase at a rate of* 4Δ*, when* Δ *increases. The increase rate probably lies between* Δ *and* 2Δ *in worst case.*

5 Analysis on Unrelated Machines

In the previous section the performance of the Greedy Algorithm on identical machines was tested and compared to algorithms. In this section we will look at the unrelated machines case, which is what the algorithm was actually designed for. This algorithm is said to be $(8+4\Delta)$-competitive. We will show whether or not we can find instances for which the performance is indeed close to this given performance bound. Next to this, we take a look at the performance of this algorithm with respect to 'average' instances. The performance of this algorithm will again be compared with the optimal value of the LP relaxation and the policy ASSIGN(X), both defined in paragraph 3.1.

5.1 Experiment

The experiment for unrelated machines has a similar structure as the experiment in paragraph 4.1. The simulation code, which was roughly described in paragraph 4.2, for the **Greedy Algorithm** and the **Linear Programming Relaxation** with the accompanying algorithm ASSIGN(X) is identical. The instances are the only thing that was changed for this part of the experiments.

Firstly, a comparison between both the Greedy Algorithm and the ASSIGN(X) algorithm with respect to the optimal LP relaxation solution are made on 'common' instances. This is an interesting experiment, since here we can see how the Greedy Algorithm performs with respect to ASSIGN(X) on instances which have not been constructed to be difficult for these algorithms. These common instances have different amounts of machines and jobs, with exponentially distributed processing times P_{ij}. The 'common' instances are defined by:

Instance 4.

$$\begin{aligned} P_{ij} &\sim \text{Exp}(k_i) && k_i \in \mathbb{Z}, 1 \leq k_i \leq 10, \forall j \in J \\ P_{ij} &= P_{mj} && \forall i, m \in M, \forall j \in J \\ w_j &= 1 && \forall j \in J \end{aligned}$$

The expected value of these processing times takes integral values from 1 upto and including 10, i.e. $0 \leq \mathbb{E}[P_{ij}] \leq 10, \mathbb{E}[P_{ij}] \in \mathbb{Z}$. Notice that $\Delta = 1$, and that the processing times P_{ij} for a job j might have different expected values for different machines i. The amount of jobs in the simulation ranges from 1 to 9 and the amount of machines from 2 to 4.

Per combination of amount of jobs and machines 100 instances, with randomly drawn values of $\mathbb{E}[P_{ij}]$, in the given domain, are used as input. Per instance the optimal values of the Greedy Algorithm, LP and LP ASSIGN(X) are stored and compared. The calculation of these values has been explained in paragraph 4.2.

After the common instances we will look at a few types of worst case instances, trying to exploit the fact that the machines are unrelated. In both instances we will construct jobs which have a low expected processing times on one half of the machines and notably worse expected processing times on the other half of the machines. The first type of instances had two types of jobs:

Instance 5.

$$\begin{aligned} P_{ij} &\sim \text{Exp}(4) && \text{for } 1 \leq j \leq |M|/2,\ \forall i \in M \\ P_{ij} &\sim \begin{cases} \text{Exp}(4),\ 1 \leq i \leq |M|/2 \\ \text{Exp}(1000),\ |M|/2+1 \leq i \leq |M| \end{cases} && \text{for } |M|/2+1 \leq j \leq |J| \\ w_j &= 1 && \forall j \in J \end{aligned}$$

The jobs are also presented to the Greedy Algorithm in this order, to ensure that the short jobs are put in the first half of the machines. After that the jobs which have a longer processing time on the second half of the machines will be put at the same first half of the machines. This results in the fact that the second half of the machines remains idle. By constructing the instance like this, we can exploit the unknowingness of the Greedy Algorithm about the jobs that still have to be presented. This instance will also be given to the LP relaxation and the accompanying $\textsc{Assign}(X)$ algorithm to obtain a comparison. The instance will be run for different values of $|M|$ and $|J|$.

Another instance which exploits a different feature of the unrelated machines model will be simulated. This instance is defined as follows:

Instance 6.

$$\begin{aligned}
&|J| = |M| && \\
&P_{ij} \sim \text{Exp}(k) && \text{for } 1 \leq j \leq |M|/2,\ \forall i \in M \\
&P_{ij} \sim \begin{cases} \text{Exp}(k),\ 1 \leq i \leq |M|/2 \\ \text{Exp}(2k-1),\ |M|/2+1 \leq i \leq |M| \end{cases} && \text{for } |M|/2+1 \leq j \leq |M| \\
&w_j = 1 && \forall j \in J
\end{aligned}$$

In this case, the jobs are presented to the Greedy Algorithm in the exact same order as defined above. This results in the fact that firstly the jobs which have the same short expected processing time on all machines are assigned to the first half of the machines. After that, the second half of the jobs will be assigned to the second half of the machines, since this results in a lower expected weighted completion time. This instance makes use of the fact that the Greedy Algorithm does not know which jobs still have to be presented. The simulation is run for different values of k and $|M|$. In the end a comparison between the Greedy Algorithm, the optimal LP solution and $\textsc{Assign}(X)$ is made.

We considered a few other instances, but these were not included in the paper. Firstly, variants on Instance 5 and 6 with different amounts of jobs or slightly different processing times have been considered. These were not included, because they behaved similarly to these instances, and hence provided no extra information. An unrelated machine variant of Instance 3 was also considered, but after a few test runs we did not gain any information that the other instances had not already given, so these instances did not make it to this paper. The last type of instances that is not included in this paper, is the one were weights w_j are chosen randomly, equal to the processing time P_{ij}, or in some other manner. For all options tried the behaviour was similar to changing the processing time P_{ij} instead of the weight w_j, so also these type of instances did not provide new information and were thus not included here.

5.2 Results & Analysis

The first instance that was simulated was the 'common' Instance 4, consisting of jobs with various exponential distributions. The results hereof can be found in Table 5 in the Appendix. In this table we see that the multiplicative gap between the Greedy Algorithm and the optimal value of the LP relaxation can become quite big. It seems however that the worst case gap will not become a lot bigger than 1.70. Not only is this the highest value we obtain in this simulations, but when the amount of jobs gets bigger, the gap seems to become smaller. Besides that, an increasing amount of machines does not seem to have a big influence. It is however difficult to be sure since the simulation could not be run on bigger instances. But it seems logical that the gap will at least not become bigger than 2.

Observation 6. *The multiplicative gap between the expected weighted completion time of the Greedy Algorithm and the optimal value of the LP relaxation is quite big. However, it will most likely not exceed 2 and certainly not come close to* $8+4\Delta$.

On the other hand, when we compare the Greedy Algorithm and $\textsc{Assign}(X)$ we observe the following:

Observation 7. *On 'common' instances with unrelated machines, the Greedy Algorithm and* $\textsc{Assign}(X)$ *have on average almost the same performance.*

Observation 8. *The worst case performance guarantee, of* $\textsc{Assign}(X)$ *on 'common' instances with unrelated machines is notably better than the worst case guarantee of the Greedy Algorithm.*

After these 'common' instances, two types of problematic instances were simulated. The first type of instance is described by Instance 5 and the results of the simulation can be found in Table 6 in the Appendix. We see here that the multiplicative gap between the LP solution and the Greedy Algorithm can get as big as 1.81, if the amount of long jobs equals the amount of machines. Besides that, we also see that the Greedy Algorithm performs a lot worse than the $\textsc{Assign}(X)$ policy, which has a worst case performance bound of approximately 1.42.

Observation 9. *For instances for unrelated machines, with jobs that have very high processing time on one machine* $\textsc{Assign}(X)$ *performs a factor* ≈ 1.4 *better than the Greedy Algorithm.*

The other difficult instance is defined by Instance 6 and the results of the simulation can be found in Table 7. For this instance it looks as if the worst-case performance bound of the Greedy Algorithm is converging to approximately 1.5. This is a better gap than we saw at the previous instance, but still worse than the average performance of the Greedy Algorithm on a 'common' instance. It is however striking that the performance of the Greedy Algorithm is a lot worse than the performance of the $\textsc{Assign}(X)$ algorithm, which returns a policy that is nearly optimal. When the expected processing time is high enough the LP relaxation works best and the multiplicative gap approaches 0, whereas the Greedy Algorithm approaches 1.5. So here we see that the Greedy Algorithm lacks in performance, where $\textsc{Assign}(X)$ works almost perfect.

Observation 10. *On 'problematic' instances for unrelated machines, where jobs have large differences in processing times for different machines,* Assign(X) *should be preferred over the Greedy Algorithm when looking at optimality of the resulting policy.*

When we combine all results, we conclude that the performance gap between the Greedy Algorithm and the LP solution can get quite big (for $\Delta = 1$). It looks as if the gap can be extended to 2, by choosing the right parameters for the jobs. However the bound of $8 + 4\Delta$ seems to be quite far out of reach.

6 Derandomization of Assign(X)

6.1 Derandomization

In previous sections we simulated the algorithm Assign(X) quite a couple of times. It turned out that the multiplicative gap between the optimal value of the LP relaxation and the Assign(X) policy can become quite big. Sometimes, this policy had even worse performance than the Greedy Algorithm, which is unexpected. This gave us enough reason to look into the derandomization of this algorithm in order to increase the performance. The technique we will use for this, is the method of conditional probabilities. This method is implicitly contained in a paper of Erdös and Selfridge [10] and was extended to general cases by Spencer [5]. Since this method uses the expected weighted completion, we need an expression using the LP solution to calculate this value in polynomial time.

Theorem 1. *The expected weighted completion time* $\mathbb{E}\left[\sum_{j\in J} w_j C_j\right]$, *for given values* $\mathbb{P}[i \to j]$ *is equal to*

$$\sum_{j\in J} w_j \sum_{i\in M} \mathbb{P}[j \to i] \sum_{k\in H(j,i)} \mathbb{E}[P_{ik}]\mathbb{P}[k \to i | j \to i].$$

Proof. Firstly, using the linearity of the expected value we obtain

$$\mathbb{E}\left[\sum_{j\in J} w_j C_j\right] = \sum_{j\in J} w_j \mathbb{E}[C_j]. \tag{7}$$

Secondly, by conditioning on the assignment of job j

$$\mathbb{E}[C_j] = \sum_{i\in M} \mathbb{P}[j \to i]\mathbb{E}[C_j | j \to i]. \tag{8}$$

Now, recall the definition of $H(j,i)$,

$$H(j,i) := \{k \in J \mid \frac{w_k}{\mathbb{E}[P_{ik}]} > \frac{w_j}{\mathbb{E}[P_{ij}]}\} \cup \{k \in J \mid k \le j,\ \frac{w_k}{\mathbb{E}[P_{ik}]} = \frac{w_j}{\mathbb{E}[P_{ij}]}\},$$

and index the jobs k in $H(j,i)$ by $k_1, k_2, k_3, \ldots$ in non-increasing order of $w_k/\mathbb{E}[P_{ik}]$, breaking ties by index. Remark that $H(j,i) \neq \emptyset$ and that job j is the last element of the ordered sequence $\{k_i\}$. Thirdly, by conditioning on jobs k_i,

$$\begin{aligned}
\mathbb{E}[C_j | j \to i] &= \mathbb{P}[k_1 \to i]\mathbb{E}[C_j | j \to i, k_1 \to i] + (1 - \mathbb{P}[k_1 \to i])\mathbb{E}[C_j | j \to i, k_1 \not\to i] \\
&= \mathbb{P}[k_1 \to i](\mathbb{E}[P_{ik_1}] + \mathbb{E}[C_j | j \to i, k_1 \not\to i]) + (1 - \mathbb{P}[k_1 \to i])\mathbb{E}[C_j | j \to i, k_1 \not\to i] \\
&= \mathbb{P}[k_1 \to i]\mathbb{E}[P_{ik_1}] + \mathbb{E}[C_j | j \to i, k_1 \not\to i] \\
&\ldots \text{ conditioning on first element of H(j,i), until } |H(j,i)| = 1 \implies H(j,i) = \{j\} \\
&= \sum_{k \in H(j,i) \setminus \{j\}} \mathbb{P}[k_1 \to i]\mathbb{E}[P_{ik_1}] \ + \ \mathbb{E}[C_j | j \to i, k \not\to i, \forall k \in H(j,i) \setminus \{j\}] \\
&= \sum_{k \in H(j,i) \setminus \{j\}} \mathbb{P}[k_1 \to i]\mathbb{E}[P_{ik_1}] \ + \ \mathbb{E}[P_{ij}] \\
&= \sum_{k \in H(j,i)} \mathbb{E}[P_{ik}]\mathbb{P}[k \to i | j \to i] \qquad (9)
\end{aligned}$$

In the final step we use $\mathbb{P}[k \to i | j \to i] = \mathbb{P}[k \to i], \ \forall k \neq j$ and $\mathbb{P}[k \to i | j \to i] = 1, \ k = j$. Finally, by combining Equations (7), (8), and (9) we obtain

$$\mathbb{E}\big[\sum_{j \in J} w_j C_j\big] = \sum_{j \in J} w_j \sum_{i \in M} \mathbb{P}[j \to i] \sum_{k \in H(j,i)} \mathbb{E}[P_{ik}]\mathbb{P}[k \to i | j \to i]. \qquad (10)$$

Hence we conclude our proof. □

Using equation (10) we can easily calculate $\mathbb{E}\big[\sum_{j \in J} w_j C_j\big]$. We can also use the same formula to calculate $\mathbb{E}\big[\sum_{j \in J} w_j C_j | j_k \to i_m\big]$, by setting $P[j_k \to i]$ to equal 1 for machine $i = i_m$ and 0 for all other machines $i \neq i_m$.

Since, we have obtained expressions for the expected weighted completion for given values $\mathbb{P}[i \to j]$, we can calculate $\mathbb{E}\big[\sum_{j \in J} w_j C_j\big]$ for all solutions X of the LP relaxation. This means we can calculate the expected weighted completion time of ASSIGN(X) using equation (10). The method of conditional probabilities states that we can now derandomize ASSIGN(X), by making a decision per job j by using equation (10).

Since $\mathbb{E}\big[\sum_{j \in J} w_j C_j\big] = \sum_{i \in M} \mathbb{P}[j_1 \to i]\mathbb{E}\big[\sum_{j \in J} w_j C_j | j_1 \to i\big]$, we know there exists a machine $i_{OPT} \in M$ such that $\mathbb{E}\big[\sum_{j \in J} w_j C_j | j_1 \to i_{OPT}\big] \leq \mathbb{E}\big[\sum_{j \in J} w_j C_j | j_1 \to i\big]$, $\forall i \in M$. But also $\mathbb{E}\big[\sum_{j \in J} w_j C_j | j_1 \to i_{OPT}\big] \leq \mathbb{E}\big[\sum_{j \in J} w_j C_j\big]$.

The method of conditional probabilities tells us that we can now subsequently assign job j to machine i_{OPT} and be sure that our expected value is less than or equal to that of the random version of ASSIGN(X). The derandomization of ASSIGN(X) is depicted by Algorithm 4.

```
Data: LP solution: $X$; Machines: $M$; Jobs: $J$
Result: Policy: $\Pi$; Policy value: $optValue$
for $j \in J$ do
    $optMachineValue = \infty$;
    $optMachine =$ None;
    for $i \in M$ do
        $machineValue = \mathbb{E}\left[\sum_{k\in J} w_k C_k | j \to i\right]$   (use Equation (10));
        if $machineValue \leq optMachineValue$ then
            $optMachineValue = machineValue$;
            $optMachine = i$;
        end
    end
    $j \to optMachine$;
    Update $\Pi$ with $(j \to i)$   (in non-increasing order of $w_j/\mathbb{E}[P_{ij}]$);
    Update $X$ to $X|(j \to i)$;
    $optValue = optMachineValue$;
end
```

Algorithm 4: Derandomized version of Assign(X).

If one looks closely at the algorithm, the complexity of the determinization of Assign(X) can be determined. In Algorithm 4 we see that the outer for-loop has complexity $O(n)$, where n is the amount of jobs. Inside this for-loop, we see another for-loop of complexity $O(m)$, with m equal to the amount of machines. So the complexity we can see in Algorithm 4 is $O(nm)$.

However, we have to take into account that on the inside of these for-loops the value $\mathbb{E}\left[\sum_{k\in J} w_k C_k | j \to i\right]$ is being calculated. For this we use Equation (10), with a slightly adapted version of the LP solution X as explained above. It is easy to see, that the complexity of calculating this sum $O(n^2 m)$. By combining all this derive the following fact:

Fact 1. *The complexity of the derandomized version of* Assign(X) *is* $O(n^3 m^2)$.

This result is desirable, since it means that our policy can be calculated in polynomial time once we have obtained the optimal LP solution X.

6.2 Analysis & Results

The performance of the derandomized version of Assign(X) has also been compared to the random version of this algorithm. This has been done using the derandomized version on the same instances on which the random version of Assign(X) has also been used (Instances 1, 4, 5 and 6). A comparison can be made by comparing the optimality gap between each version of Assign(X) and the optimal value of the LP relaxation. The simulations have been performed in the same way as explained in section 4 and 5, Instances 1, 4, 5 and 6 are also defined there. The Python code which was used for the derandomization can be found in Appendix B.

Firstly, we compare both versions of ASSIGN(X) on Instance 1, which has identical machines. We observe that the derandomized version has a better performance than the random version, both on average and in worst case. When the amount of jobs gets higher, the factor between these two algorithms is approximately 1.1. In this case it would definitely pay off to use the derandomized version.

Observation 11. *On 'common' instances with identical machines, derandomization of* ASSIGN(X) *improves both the average and worst case performance noticeably with a factor* ≈ 1.1.

On Instance 4, which is the unrelated machine version of Instance 1, we observe similar behaviour. On average the difference factor between the random and randomized version is again approximately 1.1, however when looking at the worst case, we see that the performance is almost similar.

The simulation results on Instance 5 also show a notable difference in performance between the random and derandomized version of ASSIGN(X). Here the multiplicative gap between the two algorithms also lies around 1.1, but is sometimes quite a bit bigger.

Observation 12. *On instances with unrelated machines, derandomization of* ASSIGN(X) *improves both the average and worst case performance notably with a factor* ≈ 1.1.

With Instance 6 we see something quite remarkable, namely that the derandomized version always yields the optimal policy, whereas the random version does not. This can be observed by noticing that the expected weighted completion time of this algorithm is always equal to the optimal value of the LP relaxation. Since this value is lower than or equal to the weighted completion time of an optimal schedule, the policy of the derandomized ASSIGN(X) must be optimal here.

Observation 13. *On some instances the derandomization of* ASSIGN(X) *yield an optimal policy, whereas the random version does not.*

If we now combine all results we see here, we see that derandomizing ASSIGN(X) enhances the performance greatly (on average with a factor 1.1), at the cost of a small polynomial-time computation. So it would definitely be advised to use this derandomized policy, which for some instances even yields an optimal policy.

7 Conclusions

After several simulations, with different stochastic scheduling instances, of the Online Greedy Algorithm from [4] we can conclude that worst case performance bound is most likely lower than $8+4\Delta$. The Greedy Algorithm however does not have better performance than the WSEPT rule on identical machines. The performance of the online Greedy Algorithm, is however notably better than the LP based algorithm $\textsc{Assign}(X)$ from 3.

In the unrelated machines case, we see that the performance of the online Greedy Algorithm is a bit worse than that of the online LP based algorithm $\textsc{Assign}(X)$. It should however be taken into account that the computation of $\textsc{Assign}(X)$ is computationally intractable. This is why the balance between computation and performance should be found and both algorithms can be useful in different situations. Since in the online case the Greedy Algorithm is the only one for unrelated machines, and the performance from a computational perspective is quite good, it is a very promising algorithm for this type of problems.

Finally, this paper derived a derandomization of $\textsc{Assign}(X)$ which can be computed in polynomial $O(n^3m^2)$ time. The performance of this algorithm is notably better than that of the random version and for some instances even returns an optimal policy.

Acknowledgements. This work was performed under the guidance of mentor Marc Uetz, University of Twente. The author thanks him for his great support and advise whilst this paper was being written.

References

[1] R. L. Graham, E. Lawler, J. Lenstra, and A. H. G. Rinnooy Kan, “Optimization and approximation in deterministic sequencing and scheduling: A survey”, *Annals of Discrete Mathematics*, vol. 5, pp. 287–326, 1979.

[2] H. Hoogeveen, P. Schuurman, and G. Woeginger, “Non-approximability results for scheduling problems with minsum criteria”, vol. 13, pp. 157–168, 2001.

[3] M. Skutella, M. Sviridenko, and M. Uetz, “Stochastic scheduling on unrelated machines”, *Mathematics of Operations Research*, vol. 41, no. 3, pp. 851–864, 2016.

[4] V. Gupta, B. Moseley, M. Uetz, and Q. Xie, “Stochastic online scheduling on unrelated machines”, *Integer Programming and Combinatorial Optimization*, F. Eisenbrand and J. Koenemann, Eds., to appear in 2017, Lecture Notes in Computer Science.

[5] J. Spencer, “Ten lectures on the probabilistic method”, in *CBMS–NSF Regional Conference Series in Applied Mathematics*, SIAM, Philadelphia, 1987.

[6] R. Möhring, A. S. Schulz, and M. Uetz, “Approximation in stochastic scheduling: The power of lp-based priority policies”, *Journal of the ACM*, vol. 46, pp. 924–942, 1999.

[7] C. J. Jagtenberg, U. Schwiegelshohn, and M. Uetz, “Analysis of smith’s rule in stochastic machine scheduling”, *Operations Research Letters*, vol. 41, no. 6, pp. 570–575, 2013.

[8] T. Kawaguchi and S. Kyan, “Worst case bound on an lrf schedule for the mean weighted flow-time problem”, *SIAM Journal on Computing*, vol. 15, pp. 1119–1129, 1986.

[9] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 5th ed. Springer International Publishing, 2016.

[10] P. Erdös and J. Selfridge, “On a combinatorial game”, *Journal of Combinatorial Theory Series A*, vol. 14, no. 3, pp. 298–301, 1973.

A Tables

This section of the Appendix contains the tables with the results of all simulations. It is divided into two subsections, of which the first contains the tables which are used in section 4. The second subsection contains the tables for section 5.

A.1 Section 4 tables

$\|M\|$	$\|J\|$	Average				Worst			
		WSEPT	ASSIGN(X)	Greedy	Derandomized ASSIGN(X)	WSEPT	ASSIGN(X)	Greedy	Derandomized ASSIGN(X)
2	2	1.00	1.07	1.00	1.00	1.07	1.25	1.00	1.00
2	3	1.04	1.14	1.09	1.14	1.13	1.30	1.24	1.27
2	4	1.09	1.22	1.17	1.20	1.24	1.40	1.33	1.33
2	5	1.12	1.28	1.21	1.23	1.22	1.44	1.34	1.34
2	6	1.14	1.32	1.23	1.25	1.23	1.43	1.33	1.32
2	7	1.15	1.35	1.26	1.27	1.23	1.44	1.33	1.38
2	8	1.17	1.37	1.27	1.27	1.28	1.45	1.37	1.38
2	9	1.18	1.38	1.28	1.27	1.26	1.46	1.38	1.37
3	3	1.00	1.07	1.00	1.00	1.03	1.20	1.00	1.06
3	4	1.03	1.12	1.03	1.10	1.10	1.30	1.21	1.21
3	5	1.05	1.16	1.11	1.16	1.10	1.34	1.20	1.27
3	6	1.07	1.23	1.16	1.19	1.14	1.35	1.25	1.29
3	7	1.09	1.27	1.19	1.22	1.16	1.39	1.29	1.30
3	8	1.10	1.31	1.22	1.24	1.18	1.40	1.30	1.31
3	9	1.11	1.35	1.24	1.25	1.17	1.43	1.32	1.31
4	4	1.00	1.07	1.00	1.00	1.06	1.20	1.00	1.08
4	5	1.01	1.11	1.04	1.08	1.06	1.21	1.16	1.16
4	6	1.03	1.14	1.07	1.13	1.08	1.28	1.18	1.24
4	7	1.04	1.18	1.12	1.17	1.10	1.32	1.22	1.25
4	8	1.06	1.23	1.16	1.18	1.11	1.36	1.24	1.25
4	9	1.07	1.26	1.18	1.21	1.12	1.37	1.28	1.26

Table 1: Simulation of Instance 1 using respectively the WSEPT-rule, ASSIGN(X) and the Greedy Algorithm for different amounts of jobs and machines. The numbers represent the multiplicative gap between the optimal value of the LP relaxation and the respective algorithm. Both the average and worst multiplicative gap are shown.

$\|M\|$	$\|J\|$	Average	Worst	$\|M\|$	$\|J\|$	Average	Worst
10	20	1.12	1.17	60	120	1.13	1.15
10	30	1.12	1.17	60	180	1.15	1.17
10	40	1.14	1.17	60	240	1.15	1.16
10	50	1.14	1.15	60	300	1.15	1.15
20	40	1.13	1.16	70	140	1.13	1.15
20	60	1.15	1.17	70	210	1.15	1.16
20	80	1.15	1.17	70	280	1.15	1.16
20	100	1.14	1.15	70	350	1.15	1.15
30	60	1.13	1.16	80	160	1.13	1.14
30	90	1.15	1.17	80	240	1.15	1.16
30	120	1.15	1.16	80	320	1.15	1.16
30	120	1.15	1.15	80	400	1.15	1.15
40	80	1.13	1.15	90	180	1.13	1.14
40	120	1.15	1.17	90	270	1.15	1.16
40	160	1.15	1.16	90	360	1.15	1.16
40	200	1.15	1.15	90	450	1.15	1.15
50	100	1.12	1.16	100	200	1.13	1.15
50	150	1.15	1.17	100	300	1.15	1.13
50	200	1.15	1.16	100	400	1.15	1.16
50	250	1.15	1.15	100	500	1.15	1.15

Table 2: Simulation of Instance 1 using the Greedy Algorithm for different amounts of jobs and machines. The numbers represent the multiplicative gap between the average weighted completion time of the WSEPT rule and the Greedy Algorithm. Both the average and worst multiplicative gap are shown.

$\lvert M\rvert$	$\lvert J\rvert$	WSEPT	Greedy
3	7	1.11	1.24
3	13	1.18	1.24
3	19	1.17	1.24
3	25	1.19	1.24
3	31	1.21	1.23
5	17	1.15	1.25
5	32	1.18	1.24
5	47	1.20	1.24
5	62	1.20	1.24
5	77	1.21	1.24
8	43	1.18	1.25
8	83	1.22	1.25
8	123	1.21	1.23
8	163	1.21	1.23
8	203	1.22	1.23
10	64	1.19	1.25
10	124	1.21	1.24
10	184	1.21	1.24
10	244	1.23	1.23
10	304	1.23	1.24
13	109	1.21	1.24
13	213	1.21	1.24
13	317	1.21	1.24
13	421	1.22	1.23
13	525	1.22	1.23

Table 3: Simulation of the stochastic Kawaguchi Kyan instance (Instance 2) using either the WSEPT-rule or the Greedy Algorithm. Different values of |M| and |J| were used. The numbers represent the multiplicative gap between the optimal value of the expected weighted completion time and that of the respective algorithm.

$\|J\|$ \ Δ	1	2	4	8	16	32	64
20	1.28	1.47	1.51	1.50	1.48	1.52	1.50
30	1.19	1.52	1.93	2.00	1.98	1.99	1.98
40	1.17	1.40	2.07	2.47	2.48	2.52	2.48
50	1.14	1.33	1.94	2.88	2.99	3.06	2.93
60	1.13	1.29	1.73	3.06	3.50	3.49	3.53
70	1.11	1.25	1.59	2.96	3.92	3.94	4.03
80	1.10	1.22	1.52	2.72	4.34	4.49	4.52
90	1.09	1.19	1.46	2.45	4.54	5.00	5.03
100	1.08	1.17	1.41	2.21	4.63	5.45	5.53

Table 4: Simulation of Instance 3 for different amounts of jobs and different values of Δ using the Greedy Algorithm. The numbers represent the multiplicative gap between the expected weighted completion time of the WSEPT rule and the Greedy Algorithm.

A.2 Section 5 tables

$\lvert M\rvert$	$\lvert J\rvert$	Average			Worst		
		ASSIGN(X)	Greedy	Derandomized ASSIGN(X)	ASSIGN(X)	Greedy	Derandomized ASSIGN(X)
2	2	1.09	1.08	1.00	1.50	1.50	1.00
2	3	1.17	1.16	1.08	1.53	1.53	1.16
2	4	1.23	1.24	1.12	1.36	1.55	1.25
2	5	1.27	1.29	1.17	1.41	1.60	1.34
2	6	1.30	1.31	1.17	1.47	1.60	1.40
2	7	1.32	1.35	1.20	1.49	1.63	1.45
2	8	1.32	1.35	1.21	1.42	1.58	1.41
2	9	1.34	1.37	1.24	1.55	1.57	1.52
3	3	1.13	1.11	1.00	1.53	1.53	1.00
3	4	1.19	1.19	1.10	1.44	1.52	1.23
3	5	1.22	1.23	1.20	1.49	1.76	1.35
3	6	1.25	1.28	1.24	1.42	1.55	1.32
3	7	1.29	1.33	1.28	1.41	1.65	1.32
3	8	1.32	1.36	1.29	1.50	1.69	1.39
3	9	1.34	1.38	1.31	1.47	1.63	1.45
4	4	1.15	1.14	1.00	1.49	1.42	1.13
4	5	1.19	1.20	1.08	1.35	1.63	1.15
4	6	1.25	1.25	1.16	1.53	1.47	1.27
4	7	1.27	1.28	1.23	1.43	1.59	1.37
4	8	1.30	1.32	1.28	1.52	1.70	1.42
4	9	1.31	1.34	1.29	1.49	1.63	1.48

Table 5: Simulation of Instance 4 using respectively the Greedy Algorithm and ASSIGN(X) with different amount of machines and jobs. The numbers represent the multiplicative gap between the optimal value of the LP relaxation and the expected weighted completion time of the algorithm.

$\vert M\vert$	$\vert J\vert$	Assign(X)	Greedy	Derandomized Assign(X)
2	2	1.14	1.50	1.00
2	3	1.20	1.81	1.20
2	4	1.27	1.81	1.27
2	5	1.28	1.75	1.28
2	6	1.28	1.67	1.28
4	4	1.17	1.50	1.00
4	6	1.32	1.81	1.20
4	8	1.34	1.81	1.27
4	10	1.36	1.75	1.28
4	12	1.35	1.67	1.28
6	6	1.18	1.50	1.00
6	9	1.34	1.81	1.20
6	12	1.34	1.81	1.27
6	15	1.40	1.75	1.28
6	18	1.37	1.67	1.28
8	8	1.26	1.50	1.13
8	12	1.35	1.81	1.20
8	16	1.42	1.81	1.27
8	20	1.41	1.75	1.28
8	24	1.38	1.67	1.28
10	10	1.23	1.20	1.10
10	15	1.37	1.81	1.20
10	20	1.41	1.81	1.27
10	24	1.41	1.75	1.28
10	30	1.38	1.67	1.28

Table 6: Simulation of Instance 5 using respectively the Greedy Algorithm and Assign(X) with different amount of machines and jobs. The numbers represent the multiplicative gap between the optimal value of the LP relaxation and the expected weighted completion time of the algorithm.

$\lvert M\rvert$	k	Assign(X)	Greedy	Derandomized Assign(X)
2	5	1.11	1.40	1.00
2	10	1.05	1.45	1.00
2	15	1.03	1.47	1.00
2	20	1.02	1.48	1.00
2	25	1.02	1.48	1.00
2	30	1.02	1.48	1.00
4	5	1.14	1.40	1.00
4	10	1.08	1.45	1.00
4	15	1.05	1.47	1.00
4	20	1.04	1.48	1.00
4	25	1.03	1.48	1.00
4	30	1.02	1.48	1.00
6	5	1.15	1.40	1.00
6	10	1.14	1.45	1.00
6	15	1.10	1.47	1.00
6	20	1.09	1.48	1.00
6	25	1.07	1.48	1.00
6	30	1.06	1.48	1.00
8	5	1.17	1.40	1.00
8	10	1.12	1.45	1.00
8	15	1.14	1.47	1.00
8	20	1.09	1.48	1.00
8	25	1.07	1.48	1.00
8	30	1.06	1.48	1.00
10	5	1.21	1.40	1.00
10	10	1.17	1.45	1.00
10	15	1.09	1.47	1.00
10	20	1.08	1.48	1.00
10	25	1.05	1.48	1.00
10	30	1.03	1.48	1.00

Table 7: Simulation of Instance 6 using respectively the Greedy Algorithm, Assign(X) (random and derandomized version) with different amounts of machines and short processing times k. The numbers represent the multiplicative gap between the optimal value of the LP relaxation and the expected weighted completion time of the algorithm.

B Python Code

In this section of the appendix all code used in the simulations is included. Only the programming of the several loose instances is not included. It is however quite easy to replicate the instances using the Job class, which can be found in the Problem module. Documentation of each module and the included methods and classes is included in the code.

This section of the appendix contains three modules: Main, Problem and Algorithm. The module Main contains the code for performing a simulation, the instance should also be defined here and the algorithms that should be used can be called here. Next to this,Main also contains method for the calculations involving the derandomization of ASSIGN(X).

The module Problem contains three classes Problem, Machine and Job. The former defines a stochastic scheduling problem with all its properties, by using the latter two classes. These two classes define respectively a machine and a job in a stochastic scheduling problem. Finally, the Algorithm module contains classes for executing the algorithms used in this paper.

B.1 Main module and Derandomization of ASSIGN(X)

```
"""
This is the main code of the package Scheduling of the Bachelor
↪  Assignment. Only this code should be run.
The other modules in this package shouldn't be changed.
This module also contains two methods for the derandomization
↪  of Assign(X).
"""
#      File name:   main.py
# Python version:   3.5
#         Author:   Tariq Bontekoe
#   Date created:   01-05-2017
#        Version:   24-06-2017

from algorithm import *
from math import inf
from typing import List


def higher(problem: 'Problem', job_j: 'Job', machine_i:
↪  'Machine') -> List['Job']:
    """
    This method calculates all jobs that would have higher
↪   priority than job j on  machine i, mathematically defined
    by H(j,i).
    :param problem: the problem instance which contains all
↪   jobs and machines.
    :param job_j: the job j for which H(j,i) should be
↪   calculated
    :param machine_i: the machine i on which H(j,i) is
↪   calculated
    :return: a list containing jobs which represents H(j,i)
    """
    higher_list = []
    job_j_ratio = job_j.weight /
     ↪  job_j.exp_time[machine_i.number]
    for job_k in problem.jobs:
        job_k_ratio = job_k.weight /
         ↪  job_k.exp_time[machine_i.number]
        if job_k_ratio > job_j_ratio:
            higher_list.append(job_k)
        elif job_k_ratio == job_j_ratio and job_k.number <=
         ↪  job_j.number:
            higher_list.append(job_k)
    return higher_list


def derandomize(problem: 'Problem', solution:
↪   List[List[float]]) -> Tuple[float, List[List[float]]]:
    """
    This method calculates the derandomized version of
↪   Assign(X), using LP relaxation solution X. It returns both
↪   the
    expected weighted completion time and the optimal policy.
    :param problem: The problem on which the derandomized
↪   version of Assign(X) is applied.
    :param solution: The LP relaxation solution X for the
↪   problem.
    :return: First return argument is the expected weighted
↪   completion time, second list is an adapted version of $X$
    which represents the schedule which results after this
↪   algorithm has been applied.
    """
    for job in problem.jobs:
        lowest_value = inf
        lowest_machine = None
        for machine in problem.machines:

            # build solution for this assignment
        solution_temp = solution[:]
        for i in range(len(problem.machines)):
            solution_temp[i][job.number] = 0.0
        solution_temp[machine.number][job.number] = 1.0

        # calculate expected value given j->i
        value = 0
        for job_j in problem.jobs:
            value_job = 0
            for machine_i in problem.machines:
                value_machine = 0
                for job_k in higher(problem, job_j,
                 ↪  machine_i):
                    value_machine +=
                     ↪  job_k.exp_time[machine_i.number] *
                     ↪  solution_temp[machine_i.number][job_k.number]
                value_machine *=
                 ↪  solution_temp[machine_i.number][job_j.number]
                value_job += value_machine
            value_job *= job_j.weight
            value += value_job
        if value < lowest_value:
            lowest_value = value
            lowest_machine = machine
    # best machine for this job is known
    for i in range(len(problem.machines)):
        solution[i][job.number] = 0.0
    solution[lowest_machine.number][job.number] = 1.0

# calculate final value
value = 0
for job_j in problem.jobs:
    value_job = 0
    for machine_i in problem.machines:
        value_machine = 0
        for job_k in higher(problem, job_j, machine_i):
            value_machine +=
             ↪  job_k.exp_time[machine_i.number] *
             ↪  solution[machine_i.number][job_k.number]
        value_machine *=
         ↪  solution[machine_i.number][job_j.number]
        value_job += value_machine
    value_job *= job_j.weight
    value += value_job
    return value, solution


if __name__ == '__main__':

    n_wsept = 10000
    n_lp = 10000

    machines = 0
    jobs = 0
    problem = Problem(machines)
    """
    INSERT INSTANCE HERE (CHANGE problem, jobs AND/OR machines
 ↪  (DEFINED STRAIGHT ABOVE) IF NECESSARY)
    """

    print("--- WSEPT ---")
    avg_weighted_completion_time = 0
    avg_completion_time = 0
    for t in range(1, n_wsept + 1):
        problem.reset()
        for job in problem.jobs:
            job.draw_real_times()
        weighted_completion_time, completion_time =
         ↪  WSEPT(problem).run()
        avg_weighted_completion_time +=
         ↪  weighted_completion_time
        avg_completion_time += completion_time
    print("### Total weighted completion time: {}
     ↪  ###".format(avg_weighted_completion_time / n_wsept))
    print("### Completion time: {}".format(avg_completion_time
     ↪  / n_wsept))
    wsept = avg_weighted_completion_time / n_wsept

    print("--- GREEDY ALGORITHM ---")
    problem.reset()
    greedy, *trash = GreedyAlgorithm(problem).run()

    print("--- LINEAR PROGRAMMING RELAXATION  ---")
    problem.reset()
    D = 0
    for i in range(machines):
        this_D = 0
        for job in problem.jobs:
```

```
        this_D += job.exp_time[i]
    if this_D > D:
        D = this_D
weights_sum = 0
for job in problem.jobs:
    weights_sum += job.weight
U = D * weights_sum

max_exp_time = 0
for i in range(machines):
    for job in problem.jobs:
        if job.exp_time[i] > max_exp_time:
            max_exp_time = job.exp_time[i]
R = 2 * jobs * max_exp_time

time_bound = 2 * U + R

print("Time bound: {}".format(time_bound))

lp = LinearProgram(problem, time_bound)
lp_ref = lp.solution
lp_schedule = lp.x

avg_weighted_completion_time = 0
avg_completion_time = 0
for t_lp in range(1, n_lp + 1):
    problem.reset()
    lp.assign_jobs()
    weighted_completion_time, completion_time = lp.run()
    avg_weighted_completion_time +=
     ↪  weighted_completion_time
    avg_completion_time += completion_time
print("### Total weighted completion time: {}
 ↪  ###".format(avg_weighted_completion_time / n_lp))
print("### Completion time: {}".format(avg_completion_time
 ↪  / n_lp))
lp = avg_weighted_completion_time / n_lp

print("--- DERANDOMIZATION ---")
derand, *derand_sol = derandomize(problem, lp_schedule)
print(derand_sol)
print(derand)
```

B.2 Problem module

```
"""
This module contains classes to define a stochastic scheduling
 ↪  problem, of n jobs (with possible release dates)
on m unrelated machines.
"""

#       File name:   problem.py
# Python version:   3.5
#          Author:   Tariq Bontekoe
#    Date created:   01-05-2017
#         Version:   24-06-2017

from typing import List, Tuple, Any
from queue import *
from functools import total_ordering


class ProblemError(Exception):
    def __init__(self, message):
        super().__init__(self, message)


class Problem:
    """
    Class defining a stochastic scheduling problem, containing
 ↪  jobs with stochastic processing times on unrelated
    machines.
    """

    def __init__(self, n_machines: int):
        """
        Constructor of a stochastic scheduling problem with m
 ↪  unrelated machines.
        :param n_machines: amount of machines in this problem
        """
        self._machines = tuple(Machine(i) for i in
         ↪  range(n_machines))
        self._jobs = []
        self._n_machines = n_machines
        self._n_jobs = 0

    def __repr__(self) -> str:
        return '# machines: {self._n_machines} \n' \
               '[] machines: \n {self._machines}
                ↪  \n'.format(self=self)

    @property
    def machines(self) -> Tuple['Machine', ...]:
        return self._machines

    @property
    def jobs(self) -> List['Job']:
        return self._jobs

    def add_job(self, new_job: 'Job'):
        """
        Method to add a job to this problem.
        :param new_job: Job to add to the problem.
        """
        new_job.set_number(self._n_jobs)
        self._jobs.append(new_job)
        self._n_jobs += 1

    def reset(self):
        for machine in self._machines:
            machine._queue = PriorityQueue()
        for job in self._jobs:
            job._assigned_to = None
            job._real_time = None


class Machine:
    """
    Class defining a machine for a stochastic scheduling
 ↪  problem.
    """

    def __init__(self, number: int):
        """
        Constructor of machine for a problem, with a certain
 ↪  number.
        Also gives the machine a priority queue to store the
 ↪  jobs in.
        :param number: number of this machine
        """
        self._number = number
        self._queue = PriorityQueue()

    def __repr__(self) -> str:
        return 'Machine {0} \n' \
               'Queue: {1} \n'.format(self._number,
                ↪  sorted(self._queue.queue))

    @property
    def number(self) -> int:
        return self._number

    @property
    def queue(self) -> 'Queue':
        return self._queue

    def add_to_queue(self, job: 'Job'):
        """
        This method adds a job to the queue of this machine
        :param job: job object to add to this queue
        """
        self._queue.put(job)


@total_ordering
class Job:
    """
    Class defining a job for a stochastic scheduling problem,
 ↪  having a certain stochastic processing time for m
    machines.
    """

    def __init__(self, cdf: List['Any'], real_time_f:
     ↪  List['Any'], exp_time: List[float], weight: float,
                 release_time: float = 0.0):
        """
        Constructor of a job of a problem, with a real (drawn
 ↪  from distribution) processing time, an expected one and a
        job weight.
        :param cdf: cdf function of the processing time
        :param real_time_f: function to draw a real processing
 ↪  time
        :param exp_time: expected processing time
        :param weight: weight of this job
        :param release_time: release time of this job, standard
↪   release time is 0.0
        """
        self._number = None
        self._assigned_to = None
        self._cdf = cdf
        self._real_time_f = real_time_f
        self._real_time = None
        self._exp_time = exp_time
        self._weight = weight
        self._release_time = release_time

    def __repr__(self) -> str:
        return 'Job {self._number} with weight {self._weight},
        ↪   release time {self._release_time}, exp time ' \
               '{self._exp_time} \n'.format(self=self)

    def __lt__(self, other):
        if not type(other) is Job:
            raise TypeError('unorderable types: {1} <
            ↪   {0}'.format(type(self), type(other)))
        if self._assigned_to is None or other.assigned_to is
        ↪   None:
            self_ratio = self._weight / self._exp_time[0]
            other_ratio = other.weight / other.exp_time[0]
        else:
            self_ratio = self._weight /
            ↪   self._exp_time[self._assigned_to.number]
            other_ratio = other.weight /
            ↪   other.exp_time[other.assigned_to.number]
        return self_ratio > other_ratio or (self_ratio ==
        ↪   other_ratio and self.number < other.number)

    @property
    def cdf(self) -> List['Any']:
        return self._cdf

    @property
    def real_time(self) -> List[float]:
        return self._real_time

    @property
    def exp_time(self) -> List[float]:
        return self._exp_time
    @property
    def weight(self) -> float:
        return self._weight

    @property
    def number(self) -> int:
        return self._number

    @property
    def assigned_to(self) -> 'Machine':
        return self._assigned_to

    def draw_real_times(self):
        """
        Method to draw real times from the given distribution.
        """
        self._real_time = []
        for f in self._real_time_f:
            self._real_time.append(f())

    def set_number(self, number: int):
        """
        Method for inside problem package only. Sets the number
 ↪  of this job.
        :param number: number of this job
        """
        self._number = number

    def assign(self, machine: 'Machine'):
        """
        This method assigns this job to the given machine, it
 ↪  also is put in the queue of this machine.
        :param machine: Machine to which this job is assigned.
        """
        self._assigned_to = machine
        machine.add_to_queue(self)
```

B.3 Algorithm module

```python
"""
This module contains classes for several different algorithms
↪  to solve a stochastic scheduling problem.
"""
#       File name:   algorithm.py
# Python version:   3.5
#          Author:   Tariq Bontekoe
#    Date created:   01-05-2017
#         Version:   24-06-2017

from problem import *
from typing import Tuple, List
from math import inf, ceil
from numpy.random import random
from gurobipy import *


class AlgorithmError(Exception):
    def __init__(self, message):
        super().__init__(self, message)


class Algorithm:
    """
    Class for an algorithm on a stochastic scheduling problem.
    The method which calculates the total weighted completion
↪   time E[Sum_j(w_j*C_j)] is implemented.
    """

    def __init__(self, problem: 'Problem'):
        """
        Constructor of the Algorithm abstract class.
        :param problem: Stochastic scheduling problem on which
↪   this algorithm should be run.
        """
        self._problem = problem

    def run(self) -> Tuple[float, float]:
        """
        This method calculates what the weighted completion
↪   time for the given problem was, using the Greedy Algorithm.
```

```
        :return: weighted completion time for this problem,
    ↪  completion time for this problem
        """
        if type(self) is GreedyAlgorithm:
            print("### Assignment### \n\n" +
             ↪  str(self._problem))
        weighted_completion_time = 0
        completion_time = 0
        for machine in self._problem.machines:
            queue = machine.queue
            current_machine_time = 0
            while not queue.empty():
                current_job = queue.get()
                if type(self) is GreedyAlgorithm or type(self)
                 ↪  is LinearProgram:  # use expected time for
                 ↪  solid policy
                    current_machine_time +=
                     ↪  current_job.exp_time[current_job.assigned_to.number]
                    weighted_completion_time +=
                     ↪  current_machine_time *
                     ↪  current_job.weight
                else:  # use real time (drawn) for non-solid
                 ↪  policy
                    current_machine_time +=
                     ↪  current_job.real_time[current_job.assigned_to.number]
                    weighted_completion_time +=
                     ↪  current_machine_time *
                     ↪  current_job.weight
            if current_machine_time > completion_time:
                completion_time = current_machine_time
        if type(self) is GreedyAlgorithm:
            print("### Total weighted completion time: {}
             ↪  ###".format(weighted_completion_time))
            print("### Completion time:
             ↪  {}".format(completion_time))
        return weighted_completion_time, completion_time


class GreedyAlgorithm(Algorithm):
    """
    Class for calculating the total weighted completion time
 ↪  E[Sum_j(w_j*C_j)] for the Greedy Algorithm.
    """
    def __init__(self, problem: 'Problem'):
        """
        Constructor for the Greedy Algorithm to solve this
↪   stochastic scheduling problem.
        Also assigns the jobs accordingly.
        :param problem: Stochastic scheduling problem on which
↪   this algorithm should be run.
        """
        super(GreedyAlgorithm, self).__init__(problem)
        self.assign_jobs()

    def assigned_higher_lower(self, job: 'Job', machine:
    ↪   'Machine') -> Tuple[List['Job'], List['Job']]:
        """
        Method which return as tuple of the jobs assigned to
↪   this machine that have higher/lower priority than this job.
        :param job: The job that should be assigned.
        :param machine: The machine it can be scheduled to.
        :return: Firstly a list of the higher priority jobs on
↪   this machine, secondly a list of the lower priority
        jobs on this machine.
        """
        higher = []
        lower = []
        job_ratio = job.weight / job.exp_time[machine.number]
        for other_job in self._problem.jobs:
            if other_job.assigned_to is machine:
                other_job_ratio = other_job.weight /
                 ↪   other_job.exp_time[machine.number]
                if other_job_ratio > job_ratio or
                 ↪   (other_job_ratio == job_ratio and
                 ↪   other_job.number < job.number):
                    higher.append(other_job)
                else:
                    lower.append(other_job)
        return higher, lower

    def expected_increase(self, job: 'Job', machine: 'Machine')
    ↪   -> float:
        """
        This method calculates the expected increase if a job
↪   is assigned to a machine, makes use of the
        assigned_higher_lower method.
        :param job: The job that should be assigned.
        :param machine: The machine for which the expected
↪   increase is calculated.
        :return: Expected increase of this job is assigned to
↪   this machine.
        """
        higher, lower = self.assigned_higher_lower(job,
         ↪  machine)
        job_increase = job.weight *
         ↪  (job.exp_time[machine.number] +
         ↪  sum([h.exp_time[machine.number] for h in higher]))
        lower_increase = job.exp_time[machine.number] *
         ↪  sum([l.weight for l in lower])
        return job_increase + lower_increase

    def assign_jobs(self):
        """
        This method assigns all jobs in order (defined in
↪   problem) to the machine with the lowest expected increase.
        """
        for job in self._problem.jobs:
            min_expected_increase = inf
            min_machine = None
            for machine in self._problem.machines:
                expected_increase = self.expected_increase(job,
                 ↪  machine)
                if expected_increase < min_expected_increase:
                    min_expected_increase = expected_increase
                    min_machine = machine
            job.assign(min_machine)


class LinearProgram(Algorithm):
    """
    Class for calculating the total weighted completion time
↪   E[Sum_j(w_j*C_j)] for the Linear Programming Relaxation.
    """

    def __init__(self, problem: 'Problem', time_bound: float):
        """
        Constructor for the Linear Program to solve this
↪   stochastic scheduling problem.
        Also assigns the jobs accordingly.
        :param problem: Stochastic scheduling problem on which
↪   this algorithm should be run.
```

```
        :param time_bound: Upper bound for the time, in order
↪   to bound the amount of variables.
        """
        super(LinearProgram, self).__init__(problem)
        self._time_bound = int(ceil(time_bound))
        self._solution = 0.0
        self._x = self.run_lp()

    @property
    def solution(self) -> float:
        return self._solution

    @property
    def x(self) -> List[List[float]]:
        return self._x

    def run_lp(self) -> List[List[float]]:
        # Create Model
        model = Model()

        # Create variables (standard lb = 0.0, ub = inf, type =
        ↪   continuous)
        x_vars = model.addVars(len(self._problem.machines),
        ↪   len(self._problem.jobs), self._time_bound + 1,
        ↪   name='x')

        # Set objective
        objective = LinExpr()  # construct empty linear
        ↪   objective function
        for j in range(len(self._problem.jobs)):  # build up
        ↪   the objective per job
            for i in range(len(self._problem.machines)):  # sum
            ↪   over machine
                for t in range(self._time_bound + 1):  # sum
                ↪   over time
                    coefficient = self._problem.jobs[j].weight
                    ↪   * (t +
                    ↪   self._problem.jobs[j].exp_time[i])
                    objective.addTerms(coefficient, x_vars[i,
                    ↪   j, t])
        model.setObjective(objective, GRB.MINIMIZE)  # set the
        ↪   objective function to the model

        # Set first constraint
```

```
for j in range(len(self._problem.jobs)):  # add one
 ↪  constraint per job
    constraint_one = LinExpr()  # create clean linear
     ↪  expression for job j
    for i in range(len(self._problem.machines)):  # sum
     ↪  over machine
        for t in range(self._time_bound + 1):  # sum
         ↪  over time
            constraint_one.addTerms(1.0, x_vars[i, j,
             ↪  t])
    model.addConstr(constraint_one, GRB.EQUAL, 1,
     ↪  "c_one_job{}".format(j))  # add the constraint
     ↪  to the model

# Set second constraint
for i in range(len(self._problem.machines)):
    for s in range(self._time_bound + 1):  # add one
     ↪  constraint per machine/time combination
        constraint_two = LinExpr()  # create clean
         ↪  linear expression for machine j/ time s
         ↪  combination
        for j in range(len(self._problem.jobs)):  # sum
         ↪  over jobs
            for t in range(s + 1):  # sum over time
                coefficient =
                 ↪  self._problem.jobs[j].cdf[i](s - t
                 ↪  + 1)
                if coefficient >= 10 ** (-13):  # only
                 ↪  add coefficients large enough

                     ↪  constraint_two.addTerms(coefficient,
                     ↪  x_vars[i, j, t])
            model.addConstr(constraint_two,
             ↪  GRB.LESS_EQUAL, 1.0,

                             ↪  "c_two_machine{i}_time{s}".format(i=i,
                             ↪  s=s))  # add the
                             ↪  constraint to the model

# Run model
model.setParam('OutputFlag', True)
model.optimize()
self._solution = model.getAttr(GRB.Attr.ObjVal)
```

```
        # Calculate X_ij
        x = []
        for i in range(len(self._problem.machines)):
            x_i = []
            for j in range(len(self._problem.jobs)):
                x_ij = 0
                for t in range(self._time_bound + 1):
                    if x_vars[i, j, t].getAttr(GRB.Attr.X) > 0:
                        print(i, j, t, x_vars[i, j,
                         ↪  t].getAttr(GRB.Attr.X))
                    x_ij += x_vars[i, j, t].getAttr(GRB.Attr.X)
                x_i.append(x_ij)
            x.append(x_i)

        print(x)
        # Test whether or not sums equal one
        for j in range(len(self._problem.jobs)):
            test = 0
            for i in range(len(self._problem.machines)):
                test += x[i][j]
        return x

    def assign_jobs(self):
        """
        This method assigns all jobs randomly to the right
 ↪  machine.
        """
        for job in self._problem.jobs:  # assign each job to a
         ↪  machine
            rand = random()
            upper_bound = 1
            for machine in self._problem.machines:
                upper_bound -=
                 ↪  self._x[machine.number][job.number]  #
                 ↪  bound for draw
                if rand >= upper_bound:  # this machine was
                 ↪  drawn
                    job.assign(machine)
                    break


class WSEPT(Algorithm):
    """
    Class for calculating the total weighted completion time
↪   E[Sum_j(w_j*C_j)] for the WSEPT algorithm
    """

    def __init__(self, problem: 'Problem'):
        """
        Constructor for the Greedy Algorithm to solve this
↪   stochastic scheduling problem.
        Also assigns the jobs accordingly.
        :param problem: Stochastic scheduling problem on which
↪   this algorithm should be run.
        """
        super(WSEPT, self).__init__(problem)
        self.assign_jobs()

    def assign_jobs(self):
        """
        This method assigns all jobs according to the order
↪   that would happen if WSEPT is applied.
        """
        jobs_sorted = sorted(self._problem.jobs)
        for machine in self._problem.machines:
            machine.idle_time = 0

        for job in jobs_sorted:
            first_machine_idle = self._problem.machines[0]
            first_machine_idle_time = inf
            for machine in self._problem.machines:
                if machine.idle_time < first_machine_idle_time:
                    first_machine_idle = machine
                    first_machine_idle_time = machine.idle_time
            job.assign(first_machine_idle)
            first_machine_idle.idle_time +=
             ↪  job.real_time[first_machine_idle.number]

        for machine in self._problem.machines:
            del machine.idle_time
```