Scheduling unrelated machines with job splitting, setup resources and sequence dependency

Presented as part of the MSc:
Data Science and Machine Learning

Course: Optimization in Engineering

School of Electrical and Computer Engineering

Author: Apostolos Moustakis

Student Number: 03400182

Email: apostolosmoustakis@mail.ntua.gr

Instructors (In Alphabetical Order):
V. Koumousis, N. Lagaros, A. Stamos and P. Triantafyllou

Table of Contents

List of Figures

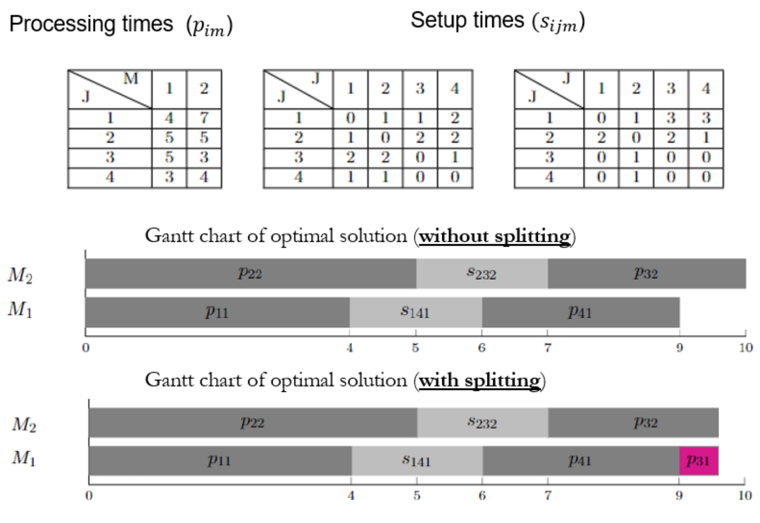
1 Introduction - Presentation of the problem

To begin with, the paper used in this assignment is written by Ioannis Avgerinos, Ioannis Mourtos, Stavros Vatikiotis and Georgios Zois and was published in July 2022, which is very recent. It can be found available at https://doi.org/10.1080/00207543.2022.2102948. The paper tackles with the problem of scheduling unrelated machines with job splitting, setup resources and sequence dependency.

The main concept of the topic is assigning jobs to multiple machines in order for them to be processed in parallel (Parallel Machine Scheduling - PMS). When jobs are processed in parallel in multiple machines the total execution time is reduced. This objective is often referred in the bibliography as minimization of makespan, which is simply the minimization of the total time for all the jobs to finish. This is also the goal of the algorithms proposed in this research paper. In similar problems there can be other objectives as well, such as tardiness minimization. Having described the main concept of assigning jobs to machines, I continue with the analysis of each term in the topic, which is presented in bullets below:

- Unrelated Machines: In the simplest form of the problem the machines are identical, which means that each job takes exactly the same time in all the machines. However, unrelated machines means that a job can have different processing time in different machines, which is more realistic.
- Job Splitting: Each job can be split in different parts and each part can be processed in different machines in parallel.
- Setup Resources: The setup time is the time needed between the execution of two jobs in the same machine. This often occurs in realistic situations as the machines need some kind of preparation before executing a specific job. In this problem the number of setups that can be performed at the same time on different machines are restricted. This also occurs in actual cases as the number of setup workers in a factory can be limited.
- Sequence Dependency: The setup times are sequence and machine dependent. Sequence dependent means that the setup time of a job i that follows job j is different from the setup time of the job j that follows job i. Furthermore, these setup times are different in different machines (machine dependent).

In order to gain a better understanding of the problem let's focus on the following example:



Processing times (p_{im})

J \ M	1	2
1	4	7
2	5	5
3	5	3
4	3	4

Setup times (s_{ijm})

J \ J	1	2	3	4
1	0	1	1	2
2	1	0	2	2
3	2	2	0	1
4	1	1	0	0

J \ J	1	2	3	4
1	0	1	3	3
2	2	0	2	1
3	0	1	0	0
4	0	1	0	0

Figure 1: Presentation of the problem: Simple Example

In this example we have 4 jobs J and 2 machines M. The processing time is denoted as p_{im}. We can see that for example job 1 has 4 processing time in machine 1 and 7 processing time in machine 2. This refers to the fact that machines are unrelated. We also have two tables with the setup times denoted as s_{ijm}. The first table refers to machine 1 and the second table to machine 2. Lets give an example: if we perform job 1 to machine 1 and then we want to perform job 3 to machine 1 the setup time is $s_{131} = 1$. However, if we want to perform the job 3 to machine 1 and then the job 1 to machine 1 the setup time is $s_{311} = 2$. This refers to the fact that the setup time is sequence dependent. Moreover, the setup time is machine dependent as in the the machine 2 we have $s_{132} = 3$, which is different than machine 1. Having described processing times and setup times we can see the optimal solution without splitting in the first Gantt chart. The total execution time of all jobs is 10. It is very simple to understand the Gannt chart: for example in the machine 1 we first process job 1 then we have the setup time for job 4 and then we process the job 4. In the second diagram we can see that the optimal solution becomes better if we split job 3 to both the machines. In this simple example we assumed that the number of setups that can be done simultaneously is at least 2.

Before continuing with the description of the methods and algorithms used for tackling this specific problem from the authors of the paper it is important to mention some additional useful information. First of all, this paper aims to fill the literature gap that exists of solving the Parallel Machine Scheduling problem assuming all of the following that are already described: unrelated machines, job splitting, setup resources and sequence dependent setup times. A lot of research is being done with variations of the problem but in simpler forms with less constraints. This specific problem is computationally expensive as the job splitting makes the solution space massive and the setup constraints demand a high number of variables. Furthermore, the methods and algorithms described in the paper are used in a real environment and specifically in a textile production process called weaving. The parallel unrelated machines are called looms and each machine takes different time to process a different job. The jobs are orders and can be split into parts, while each job refers to the production of a certain type of fabric. The number of setups that can be done at the same time on different machines is restricted due to the limited number of setup workers.

2 Implemented Algorithms and Results

This paper tackles the problem in two different steps. The first step involves the construction of lower bounds with relaxations of the problem and the creation of a Greedy Heuristic Algorithm. The lower bounds can serve as evaluation solutions to the problem. The second step is constructing an exact method and specifically Logic Based Benders Decomposition. The idea behind the whole approach is that someone can use the solution of the Greedy Heuristic (which is fast and efficient) as a warm up solution and a tight upper bound and then further improve it with the exact method (exact methods guarantee optimal solutions or very close to optimal). Using just the exact method from the beginning is computationally expensive and does not produce results for large instances of the problem. Below I describe the two steps and the results obtained in the study.

2.1 Lower bounds and Greedy Heuristic Algorithm

The lower bounds can be derived from 3 Mixed Integer Program formulations (MIPs) by relaxing the initial problem. MIPs are used to solve optimization problems that contain both integer and continuous values. Known algorithms can be used in order to solve MIPs such as Branch and Bound. Unfortunately we cannot use MIPs to solve the problem as it is very computationally expensive and has a very large number of variables. A Monolithic MILP can only solve the problem optimally for very few jobs. However, we use MIPs in relaxed instances of the problem to produce lower bounds and then evaluate the solution regarding these lower bounds. The model parameters and the variables are displayed in the two tables below:

Model parameters	Description
J	The set of jobs – including a dummy job 0
J^*	The set of jobs – excluding the dummy job 0
M	The set of machines
p_{im}	The processing time of $i \in J$ on $m \in M$
s_{ijm}	The setup time of $j \in J$ succeeding job $i \in J - \{j\}$ on $m \in M$
$s_0 \in \mathbb{R}$	The setup time of the first job, $\forall m \in M$
R	A setup resource constraint to indicate that, at each time interval, at most R machines can be set up in parallel
δ	A positive constant for (MIP 1), with $\delta \to \infty$

Table 1: Model Parameters and Description

Model variables	Description
$z_{i,m}$	1 if $i \in J^*$ is the first job to be assigned on machine $m \in M$, 0 otherwise
$f_{i,m}$	the percentage of the first $i \in J^*$ to be processed on machine $m \in M$
$y_{i,m}$	1 if $i \in J^*$ is assigned on machine $m \in M$, 0 otherwise
x_{ijm}	1 if $j \in J$ succeeds $i \in J^* - \{j\}$ on machine $m \in M$, 0 otherwise
w_{im}	the percentage of job $i \in J^*$ to be processed on machine $m \in M$
$W_{im} \in \mathbb{Z}^+$	the integer percentage of job $i \in J^*$ to be processed on machine $m \in M$
$n_{im} \in \mathbb{Z}^+$	auxiliary integer variables that ensure the feasibility of sequencing
$C_{max} \in \mathbb{R}^+$	the makespan of the schedule
$\mu_{im} \in \mathbb{R}^+$	interval variables that indicate the time interval of setup of job $i \in J^*$ machine $m \in M$

Table 2: Model Variables and Description

MIP1 and MIP2 formulations are a relaxed version of the problem as they neglect the resource constraints and assume that the setup times are constant and set to their minimum values. By solving MIP1 or MIP2 a lower bound to the optimal solution is guaranteed. MIP3 is similar to MIP1 with the main difference that jobs are not distinguished based on whether they are assigned first. Therefore, it provides a lower bound solution only if the setup time of the first job is the largest among all. MIP1, MIP2 and MIP3 formulations are displayed below.

MIP1: The objective of MIP1 is C_{max} subject to the constraints:

$y_{i,m} \geq w_{im}\ \forall i \in J^*, m \in M$
$z_{i,m} \geq f_{im}\ \forall i \in J^*, m \in M$
$z_{i,m} + y_{i,m} \leq 1\ \forall i \in J^*, m \in M$
$z_{i,m} - f_{i,m} \leq 1 - \delta\ \forall i \in J^*, m \in M$
$\sum_{m \in M} z_{i,m} + f_{i,m} = 1\ \forall i \in J^*$
$\sum_{i \in J^*} z_{i,m} \leq 1\ \forall m \in M$
$\sum_{i \in J^*} z_{i,m} \geq \sum_{i \in J} y_{i,m}\ \forall m \in M$
$\sum_{j \in J^*} (f_{j,m} \dot{p}_{j,m} + s_0 \dot{z}_{j,m} + w_{j,m} \dot{p}_{j,m} + y_{j,m} \dot{m}in_{i \in J^* - \{j\}} s_{i,j,m} \leq C_{max}\ \forall m \in M$
$y_{i,m}, z_{i.m} \in \{0,1\}, f_{i,m}, w_{i.m} \in [0,1], C_{max} \in \mathbb{R}^+, \forall i \in J^*, m \in M$

MIP2: The objective of MIP2 is minimizing C_{max} subject to the constraints:

$$\sum_{i \in J^*} w_{i,m} \dot{p}_{i,m} \leq C_{max}\ \forall m \in M, \quad \sum_{m \in M} w_{i,m} = 1\ \forall i \in J^*$$

MIP3: The objective of MIP3 is minimizing C_{max} subject to the constraints:

$$\sum_{j \in J^*} (w_{j,m} \dot{p}_{j,m} + y_{j,m} \dot{m}in_{i \in J^* - \{j\}} s_{i,j,m} \leq C_{max}\ \forall m \in M, \quad \sum_{m \in M} w_{i,m} = 1\ \forall i \in J^*$$

The Greedy Heuristic algorithm is displayed below:

Algorithm 1 GHA: A three-stage greedy heuristic

```
max_assgn ← $|J|\dot{|}M|$, $C_{max} \to \infty$
while max_assgn ≥ $|J|$ do
    Solve (MIP 1) or (MIP 2) or (MIP 3) and let $y$ be the number of job parts assigned over all machines
    max_assgn ← $y$
    Run Sequencing Stage
    Run Resource Management Stage
    Let $ldm$ be the load of each machine $m \in M$
    If necessary update $C_{max}$ with $max_{m \in M}\{ldm\}$
    max_assgn ← max_assgn − 1
end while
Return $C_{max}$
```

The Greedy Heuristic Algorithm (GHA) has 3 stages. The first stage is the Assignment stage, where MIP1, MIP2 or MIP3 is solved and jobs are splitted into parts and assigned to machines. In any of the formulations (MIP1, MIP2 or MIP3) the following constraints are added:

$\sum_{i\in J}\sum_{m\in M}(y_{i,m}+z_{i,m}) \leq \text{max\_assgn}\ \forall m \in M, \quad w_{j,m}\dot{p}_{j,m} \geq min_{i\in J^*-\{j\}} s_{i,j,m}\ \forall i \in J^*, m \in M$

The second stage is called the sequencing stage where sequences of the job parts are created on each machine. The order of the assigned job parts occurs by solving an asymmetric TSP, where the nodes are the job parts and the node distances are the sum of sequence-dependent setup time and the processing time. Lastly, in the third and final stage that is called Resource Management stage appropriate time windows are created ensuring that the resource constraints are met. More specifically, for each machine (in decreasing order of load) the earliest time that a job part can start its setup is computed, while respecting the order of job parts from the sequencing stage. Which one of MIP1, MIP2 and MIP3 is the best regarding the first stage can be determined only computationally, but it is shown in the paper that MIP3 provides the best balanced solutions regarding quality and time.

2.2 Exact method: Logic Based Benders Decomposition

Benders Decomposition is a primal-dual method that can provide an exact solution to the problem. The main components of the method are the master problem M that provides lower bounds, the subproblem S that provides upper bounds and a set of optimality and feasibility cuts that pass from S to M. The idea is to move difficult restrictions to S after solving M easily (a relaxed version of the problem), which is simple. However, we need to be cautious while transferring "knowledge" from S to M in the following iteration. This process is repeated with the expectation that the optimality gap will converge quickly. Logic-Based Benders Decomposition (LBBD) is just an extension of the original Benders Decomposition that constructs valid optimality and feasibility cuts for integer subproblems.

In this specific problem the Master problem M refers to the assignment and the sequencing stage, while the subproblem S refers to the resource management stage. During the master problem M we assume that the resources are unlimited. The jobs are split into parts and are assigned and sequenced to machines. In this way the complexity of the problem is significantly reduced. On the other hand, the subproblem S adds intervals of idle time between jobs so that the available resources are not exceeded in the sequence provided by M. It is important to note that LBBD is not used directly to the problem: a solution of the Greedy Heuristic Algorithm is computed and is used as a warm start. This provides M with a reduced tighter upper bound and helps in evaluating solutions in a reduced domain of candidate solutions. In this way LBBD coupled with the GHA can solve way larger instances of the problem (instances with up to 200 jobs and 20 machines near optimality). The whole process is described in the image below:

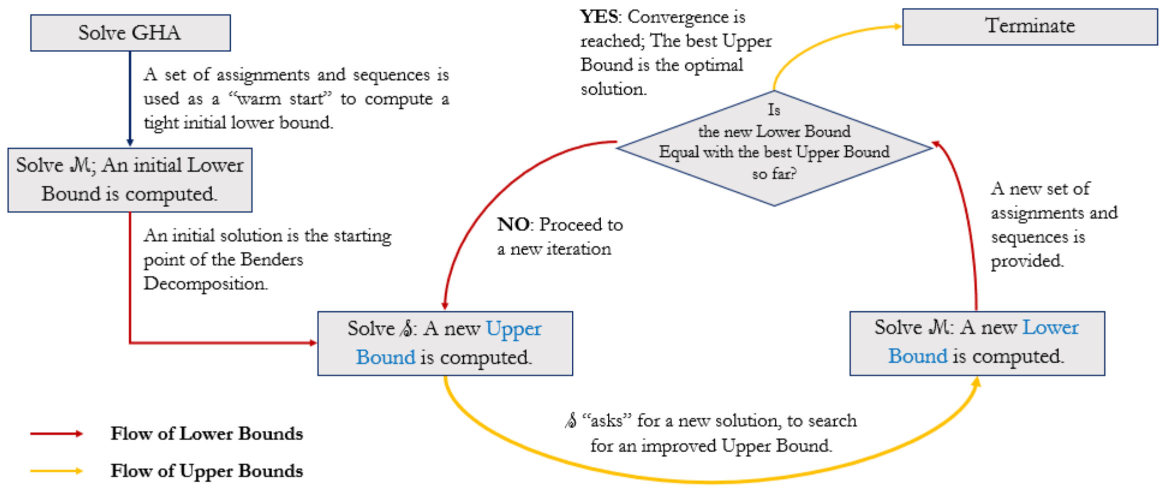

Solve GHA
A set of assignments and sequences is used as a "warm start" to compute a tight initial lower bound.
Solve $\mathcal{M}$; An initial Lower Bound is computed.
An initial solution is the starting point of the Benders Decomposition.
Solve $\mathcal{S}$: A new Upper Bound is computed.
YES: Convergence is reached; The best Upper Bound is the optimal solution.
Terminate
Is the new Lower Bound Equal with the best Upper Bound so far?
NO: Proceed to a new iteration
A new set of assignments and sequences is provided.
Solve $\mathcal{M}$: A new Lower Bound is computed.
$\mathcal{S}$ "asks" for a new solution, to search for an improved Upper Bound.
Flow of Lower Bounds
Flow of Upper Bounds

Figure 2: LBBD with GHA solution as a warm start

Let's see the following example in order to understand better the optimality cuts and the iterations that occur between S and M.

M provides S with an initial solution. This solution is a sequence of jobs without the resource constraints and thus it is considered a Lower Bound. Then S provides the correct solution to that sequence by adding the resource constraints. This implies an Upper Bound to the optimal solution. This can be seen in the image below:

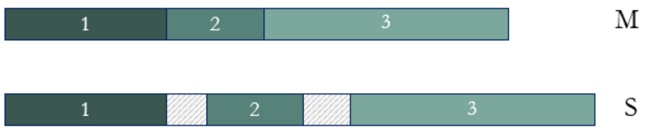


Figure 3: Initial Lower Bound of M and Upper Bound of S

In the following iterations M provides S with a new Lower Bound. This new Lower Bound can be worse than the previous one but can lead to better solutions (improved Upper Bound). This can be seen in the image below. Previously we had the sequence of jobs 1,2,3 and now he have the sequence of jobs 2,3,1. 2,3,1 is a worse Lower Bound but the new Upper Bound after the addition of the resource constraints is better (it reduces the total makespan).

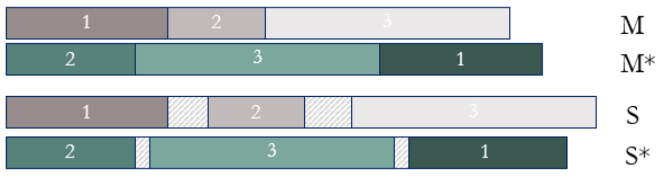
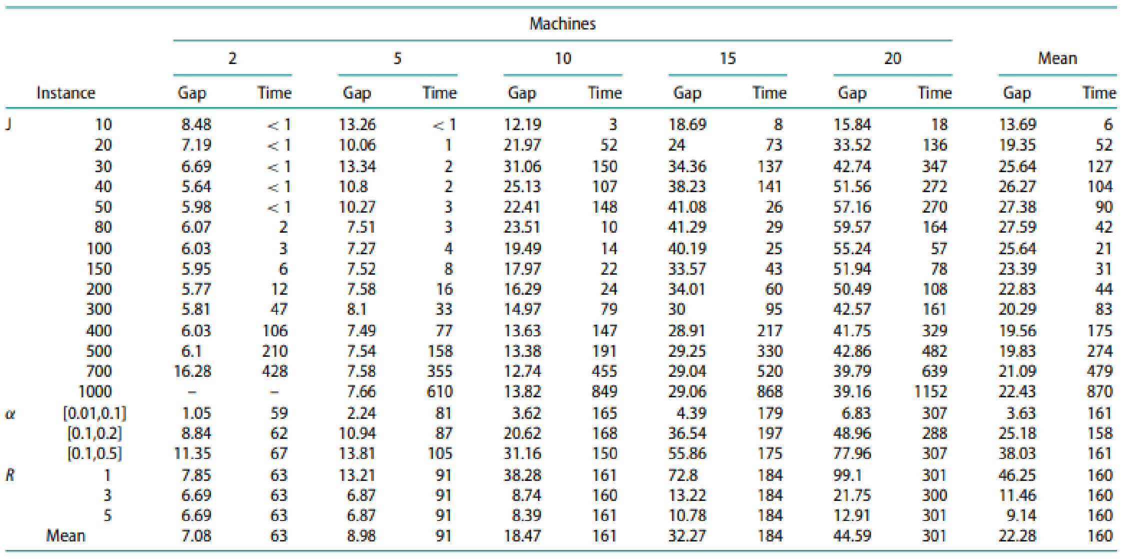


Figure 4: Iterations between M and S

The knowledge that is provided from S to M is called Optimality Cuts. After a sufficient number of iterations the new Lower Bound will be equal to the best Uper Bound (regarding the sequence of the jobs). When this happen the algorithm has converged and the best Upper Bound is the optimal solution to the problem. The Lower Bound that M will provide in the last iteration will be exactly the same with the previous iteration, making the previous Upper Bound the best one and the optimal solution.

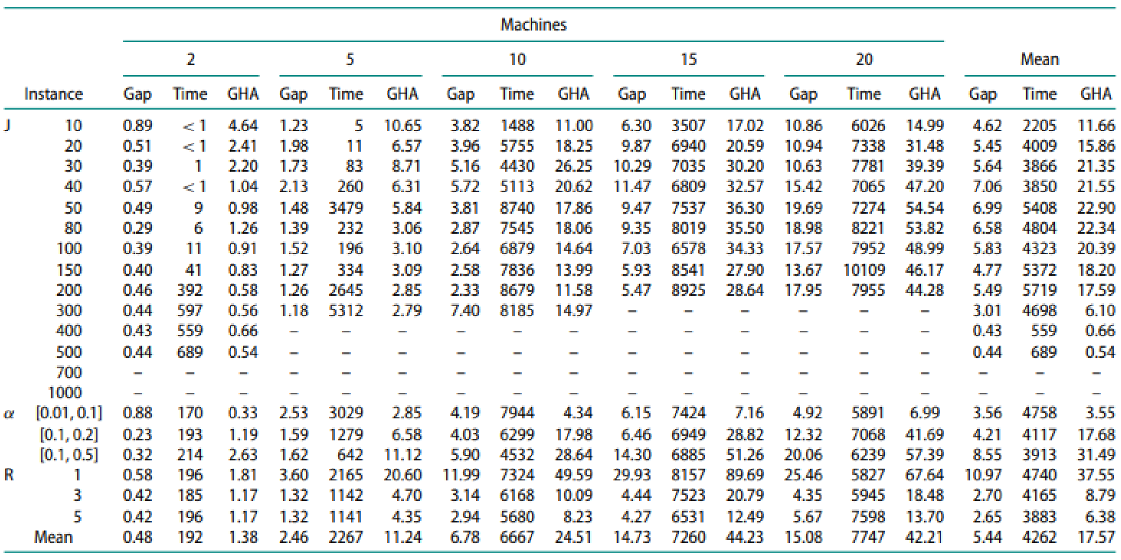
2.3 Results

The evaluation of both the Greedy Heuristic Algorithm (GHA) and the Logic Based Benders Decomposition (LBBD) takes place on benchmark instances. The results of GHA and LBBD (with the use of the GHA solution as a warm start) are presented in the tables below:



		Machines											
		2		5		10		15		20		Mean	
	Instance	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time
J	10	8.48	< 1	13.26	< 1	12.19	3	18.69	8	15.84	18	13.69	6
	20	7.19	< 1	10.06	1	21.97	52	24	73	33.52	136	19.35	52
	30	6.69	< 1	13.34	2	31.06	150	34.36	137	42.74	347	25.64	127
	40	5.64	< 1	10.8	2	25.13	107	38.23	141	51.56	272	26.27	104
	50	5.98	< 1	10.27	3	22.41	148	41.08	26	57.16	270	27.38	90
	80	6.07	2	7.51	3	23.51	10	41.29	29	59.57	164	27.59	42
	100	6.03	3	7.27	4	19.49	14	40.19	25	55.24	57	25.64	21
	150	5.95	6	7.52	8	17.97	22	33.57	43	51.94	78	23.39	31
	200	5.77	12	7.58	16	16.29	24	34.01	60	50.49	108	22.83	44
	300	5.81	47	8.1	33	14.97	79	30	95	42.57	161	20.29	83
	400	6.03	106	7.49	77	13.63	147	28.91	217	41.75	329	19.56	175
	500	6.1	210	7.54	158	13.38	191	29.25	330	42.86	482	19.83	274
	700	16.28	428	7.58	355	12.74	455	29.04	520	39.79	639	21.09	479
	1000	–	–	7.66	610	13.82	849	29.06	868	39.16	1152	22.43	870
α	[0.01,0.1]	1.05	59	2.24	81	3.62	165	4.39	179	6.83	307	3.63	161
	[0.1,0.2]	8.84	62	10.94	87	20.62	168	36.54	197	48.96	288	25.18	158
	[0.1,0.5]	11.35	67	13.81	105	31.16	150	55.86	175	77.96	307	38.03	161
R	1	7.85	63	13.21	91	38.28	161	72.8	184	99.1	301	46.25	160
	3	6.69	63	6.87	91	8.74	160	13.22	184	21.75	300	11.46	160
	5	6.69	63	6.87	91	8.39	161	10.78	184	12.91	301	9.14	160
	Mean	7.08	63	8.98	91	18.47	161	32.27	184	44.59	301	22.28	160

Figure 5: Results of Greedy Heuristic Algorithm on benchmark instances



		Machines																	
		2			5			10			15			20			Mean		
	Instance	Gap	Time	GHA	Gap	Time	GHA	Gap	Time	GHA	Gap	Time	GHA	Gap	Time	GHA	Gap	Time	GHA
J	10	0.89	< 1	4.64	1.23	5	10.65	3.82	1488	11.00	6.30	3507	17.02	10.86	6026	14.99	4.62	2205	11.66
	20	0.51	< 1	2.41	1.98	11	6.57	3.96	5755	18.25	9.87	6940	20.59	10.94	7338	31.48	5.45	4009	15.86
	30	0.39	1	2.20	1.73	83	8.71	5.16	4430	26.25	10.29	7035	30.20	10.63	7781	39.39	5.64	3866	21.35
	40	0.57	< 1	1.04	2.13	260	6.31	5.72	5113	20.62	11.47	6809	32.57	15.42	7065	47.20	7.06	3850	21.55
	50	0.49	9	0.98	1.48	3479	5.84	3.81	8740	17.86	9.47	7537	36.30	19.69	7274	54.54	6.99	5408	22.90
	80	0.29	6	1.26	1.39	232	3.06	2.87	7545	18.06	9.35	8019	35.50	18.98	8221	53.82	6.58	4804	22.34
	100	0.39	11	0.91	1.52	196	3.10	2.64	6879	14.64	7.03	6578	34.33	17.57	7952	48.99	5.83	4323	20.39
	150	0.40	41	0.83	1.27	334	3.09	2.58	7836	13.99	5.93	8541	27.90	13.67	10109	46.17	4.77	5372	18.20
	200	0.46	392	0.58	1.26	2645	2.85	2.33	8679	11.58	5.47	8925	28.64	17.95	7955	44.28	5.49	5719	17.59
	300	0.44	597	0.56	1.18	5312	2.79	7.40	8185	14.97	–	–	–	–	–	–	3.01	4698	6.10
	400	0.43	559	0.66	–	–	–	–	–	–	–	–	–	–	–	–	0.43	559	0.66
	500	0.44	689	0.54	–	–	–	–	–	–	–	–	–	–	–	–	0.44	689	0.54
	700	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	1000	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
α	[0.01, 0.1]	0.88	170	0.33	2.53	3029	2.85	4.19	7944	4.34	6.15	7424	7.16	4.92	5891	6.99	3.56	4758	3.55
	[0.1, 0.2]	0.23	193	1.19	1.59	1279	6.58	4.03	6299	17.98	6.46	6949	28.82	12.32	7068	41.69	4.21	4117	17.68
	[0.1, 0.5]	0.32	214	2.63	1.62	642	11.12	5.90	4532	28.64	14.30	6885	51.26	20.06	6239	57.39	8.55	3913	31.49
R	1	0.58	196	1.81	3.60	2165	20.60	11.99	7324	49.59	29.93	8157	89.69	25.46	5827	67.64	10.97	4740	37.55
	3	0.42	185	1.17	1.32	1142	4.70	3.14	6168	10.09	4.44	7523	20.79	4.35	5945	18.48	2.70	4165	8.79
	5	0.42	196	1.17	1.32	1141	4.35	2.94	5680	8.23	4.27	6531	12.49	5.67	7598	13.70	2.65	3883	6.38
	Mean	0.48	192	1.38	2.46	2267	11.24	6.78	6667	24.51	14.73	7260	44.23	15.08	7747	42.21	5.44	4262	17.57

Figure 6: Results of Logic Based Benders Decomposition on benchmark instances

The instances are generated randomly with numbers of machines $|M| \in \{2, 5, 10, 15, 20\}$ and numbers of jobs $|J| \in \{10, 20, 30, 40, 50, 80, 100, 200, 300, 400, 500, 700, 1000\}$ and thus we have 70 combinations. Moreover, we have that the processing time is $p_{im} = b_i \cdot a_{im} + u_{im}$, where u_{im} is selected u.a.r from [0,10], while b_i, a_{im} are selected u.a.r from [1,10] and the setup times are $s_{ijm} = p_{jm} \cdot a_{ijm}$, where $a_{ijm} \in \{[0.01, 0.1], [0.1, 0.2], [0.1, 0.5]\}$ and thus the instances are $70 \cdot 3 = 210$. Lastly, the number of working groups are $R \in \{1, 3, 5\}$ and thus the total number of experiments are $3 \cdot 210 = 630$.

The evaluation takes place regarding the execution time (the processing power plays an important role) and the Gap. Gap is equal to: $\frac{Solution - Lower\ Bound}{Lower\ Bound}$. I will thoroughly analyze the results during the implementation of Simulated Annealing. With a first look LBBD guarantees better solutions as the total average Gap is 5.44%, while the total average Gap of GHA is 22.28%. However, the average time of GHA is just a few minutes (160 seconds), while the average time of LBBD is over an hour (4262 seconds). Lastly, LBBD cannot solve very large instances of the problem (for instance with 700 and 1000 jobs) even with the GHA as a worm start.

3 Simulated Annealing

The Simulated Annealing algorithm is a metaheuristic optimization algorithm used to find the global minimum or maximum of a function. The algorithm's name comes from annealing in metallurgy, which is a technique that involves heating and controlled cooling of a material in order to change its physical properties.

When using the algorithm, the optimization problem is modeled as a system, which has some energy function $E(x)$. The algorithm starts with an initial solution x_c, which has energy E_c. The algorithm then iteratively explores possible perturbative solutions. If the new solution is better (leads the system to a lower energy), then it is accepted. However, if it is worse (leads the system to a higher energy), then it is not always rejected. Instead, it is accepted with a certain probability, which is calculated as:

$$p = exp(-\frac{E_{proposed} - E_c}{T_c})$$

Specifically, a pseudorandom number x is generated from the uniform distribution in the interval $[0, 1]$ and the solution is accepted if the above probability is greater than the pseudorandom number, otherwise it is rejected. The parameter T_c, which appears in the probability formula, is called the temperature and it decreases gradually from iteration to iteration of the algorithm. In this way the algorithm after some iterations becomes more greedy and accepts worse solutions with lower probability. Simulated Annealing has the potential to lead to better solutions as by accepting sometimes worse solutions it can escape from local optima and find the global optimum. The pseudocode for Simulated Annealing is displayed below (x_c is the initial solution):

Algorithm 2 Pseudocode of Simulated Annealing

```
Require: define DECREASE(T), define PERTURB(x)
  Begin
  $T \leftarrow initial\_temp$
  $x \leftarrow x_c$
  while $T > Final\_T$ do
      $Equilibrium \leftarrow False$
      while Equilibrium == False do
          $x\_proposed \leftarrow PERTURB(x)$
          if $E_{proposed} < E_c$ then
              $x \leftarrow x_{proposed}$
          else if $exp(-(E_{proposed} - E_c)/T_c) > Random(0,1)$ then
              $x \leftarrow x_{proposed}$
          else if then
              $x \leftarrow x$
          end if
          $step \leftarrow step + 1$
          $T \leftarrow DECREASE(T)$
      end while
  end while
  End
```

4 Implementation of Simulated Annealing and Results

In this chapter I will analyze the implementation of Simulated Annealing in this specific problem and provide the relative code. The code is written in Python with the use of Google Colab and is attached. I will provide code snippets with the implementation and explanation of the code. Briefly, the goal of this implementation is to improve the solutions provided by the Greedy Heuristic Algorithm (GHA) with the use of Simulated Annealing (SA).

As already described in the pseudocode of Simulated Annealing we mainly need to define how to decrease the temperature and how to generate perturbative solutions. We have the following:

- Temperature: Tc is the initial temperature, Tcry is the final temperature and q is the cooling factor. In each iteration the temperature is decreased regarding the cooling factor q (Tc = q · Tc). I perform experiments with combinations of Tc 200 or 500, Tcry 1 or 20 and q 0.90 or 0.95. These values are used based on bibliography and experiments.

- Perturbative Solutions: For each temperature the SA performs an internal swap first: randomly chooses 1 machine and 2 jobs in that machine (again randomly) and swaps them. In the next iteration it performs an external swap, where it chooses randomly two machines and 1 job in each machine (again randomly) and swaps them and so on. Of course every time the makespan of the perturbative solution is calculated. The makespan is considered the "Energy of the System".

Below I provide the whole code along with further explanation:

```python
import random
import time
import math
import numpy as np
import copy
import json
import csv
import pandas as pd
```

Python Code Snippet 1: Imports needed for the project

The first python code snippet is very simple and just contains the necessary imports for the project. For example, in order to be able to read the json files with the instances and the solutions of the GHA the import json was used.

```python
def simulated_annealing(process, setups, loom_seq, assignments, Tcry, Tc, q, IT, lb,
    num_machines, num_orders):
    start_time = time.time()

    # calculate initial solution

    current_assignments = copy.deepcopy(assignments)
    current_seq = copy.deepcopy(loom_seq)
    current_obj, current_tables = sol_calc(setups, loom_seq, assignments, num_machines, num_orders)

    best_sol = {'Obj': current_obj, 'Seq': current_seq, 'Sol': current_assignments}
    interior_swap = 1
    while Tcry < Tc:
        for it in range(IT):
            # create new solution
            new_assignments = copy.deepcopy(current_assignments)
            new_seq = copy.deepcopy(current_seq)
            if interior_swap == 1:
                random_machine = random.randint(0, num_machines - 1)
                if len(new_seq[str(random_machine)]) > 3:
                    job1_pos = random.randint(1, len(loom_seq[str(random_machine)]) - 1)
                    job2_pos = random.choice(
                        list(range(1, job1_pos)) + list(range(job1_pos + 1, len(loom_seq[str(random_machine)]) - 1)))
                    new_seq[str(random_machine)][job1_pos], new_seq[str(random_machine)][job2_pos] = \
                    new_seq[str(random_machine)][job2_pos], new_seq[str(random_machine)][job1_pos]
                    interior_swap = 0
                elif len(new_seq[str(random_machine)]) > 2:
                    job1_pos = 1
                    job2_pos = 2
                    new_seq[str(random_machine)][job1_pos], new_seq[str(random_machine)][job2_pos] = \
                    new_seq[str(random_machine)][job2_pos], new_seq[str(random_machine)][job1_pos]
                    interior_swap = 0
                else:
                    interior_swap = 0
            else:
                if num_machines == 2:
                    random_machine_1 = 0
                    random_machine_2 = 1
                else:
                    random_machine_1 = random.randint(0, num_machines - 1)
                    random_machine_2 = random.choice(
                        list(range(0, random_machine_1)) + list(range(random_machine_1 + 1, num_machines - 1)))
                random_job_1 = random.choice(list(new_seq[str(random_machine_1)][1:]))
                job1_pos = new_seq[str(random_machine_1)].index(random_job_1)
                random_job_2 = random.choice(list(new_seq[str(random_machine_2)][1:]))
                job2_pos = new_seq[str(random_machine_2)].index(random_job_2)
                #print (new_seq)
                #print (random_job_1)
                #print (random_job_2)
                new_seq[str(random_machine_1)][job1_pos], new_seq[str(random_machine_2)][job2_pos] = \
                new_seq[str(random_machine_2)][job2_pos], new_seq[str(random_machine_1)][job1_pos]
                new_assignments[random_job_1, random_machine_2] = process[random_job_1,random_machine_1]
                new_assignments[random_job_1,random_machine_1] = 0.0
                new_assignments[random_job_2, random_machine_1] = process[random_job_2, random_machine_2]
                new_assignments[random_job_2, random_machine_2] = 0.0

                interior_swap = 1


            new_obj, new_tables = sol_calc(setups, new_seq, new_assignments, num_machines, num_orders)
            # print (current_obj, new_obj)
            try:
                Pacc = math.exp(round((current_obj - new_obj) / Tc, 2))
            except:
                Pacc = -1
            if best_sol['Obj'] > new_obj:
                best_sol['Obj'] = copy.deepcopy(new_obj)
                best_sol['Seq'] = copy.deepcopy(new_seq)
                best_sol['Sol'] = copy.deepcopy(new_assignments)

            if new_obj < current_obj or Pacc > random.random():
                current_obj = copy.deepcopy(new_obj)
                current_seq = copy.deepcopy(new_seq)
                current_assignments = copy.deepcopy(new_assignments)

        Tc = q * Tc

    sa_gap = round((best_sol['Obj'] - lb) / lb * 100,3)
    sa_time = round(time.time() - start_time,3)
    sa_sols = {'Time': sa_time, 'Gap': sa_gap, 'Obj': best_sol['Obj']}
    return sa_sols


def sol_calc(setups, loom_seq, assignments, num_machines, num_orders):
    tables = {}
    machine_load = {}
    for m in range(num_machines):  # number of machines
        machine_load[m] = 0
        tables[m] = np.full((num_orders + 1, 4), -1.0)
        for j in loom_seq[str(m)]:
            pos_j = loom_seq[str(m)].index(j)
            tables[m][j, 0] = machine_load[m]
            tables[m][j, 1] = float(setups[m][loom_seq[str(m)][pos_j - 1]][[loom_seq[
    str(m)][pos_j]]])
            tables[m][j, 2] = assignments[j, m]
            tables[m][j, 3] = tables[m][j, 0] + tables[m][j, 1] + tables[m][j, 2]
            machine_load[m] = tables[m][j, 3]
    #print (max(machine_load.values()))
    return max(machine_load.values()), tables
```

Python Code Snippet 2: Simulated Annealing and Solution Calculation Functions

The second python code snippet contains the functions simulated_annealing and sol_calc. A brief explanation of what exactly these functions do is the following:

- simulated_annealing: The arguments of this function are the process times, setups, loom sequence, assignments, initial temperature, final temperature, cooling factor, iterations, lower bound, number of machines and number of orders (jobs). Firstly, I calculate the initial solution, which is the solution provided by the GHA. Then while the final temperature is smaller than the initial and each time for IT iterations (50 or 100) I firstly perform an interior swap (choose 1 machine randomly and then 2 jobs in that machine randomly and swap) and in the next iteration an external swap (choose 2 machines randomly and 1 job in each machine randomly and swap) and so on. This is achieved by changing the variable interior_swap to either 0 or 1 at the end of each iteration. More specifically, regarding the interior_swap if the jobs are more than 3 then 2 of them are randomly chosen and the sequence is updated, while if the jobs are more than 2 (exactly 3) then the job 1 is in position 1 and the job 2 is in position 2 and I swap them. Note that the first job is always the dummy job 0 that does not play any role. At the end I set the interior_swap to 0 in order to perform an exterior swap in the next iteration. Regarding the exterior swap I firstly pick a random machine (if the machines are 2 the choice is obvious) and then a random job in each machine and swap them. When this is done I update the sequence and the assignment table, while I also set the interior_swap to 1 to perform an interior swap in the next iteration. At the end of each iteration I calculate the new solution with the use of the function sol_calc(). If the solution is better (shorter makespan) I update the dictionary best_sol that keeps the best solution so far. Furthermore, if the solution is better or worse with acceptance probability greater than a random number generated between 0 and 1, I update the current solution. Note that the calculation of Pacc sometimes produces an error and in that case I set Pacc = -1, which means that I continue with the next iteration if the solution is worse. When the IT iterations finish I decrease the temperature regarding the cooling factor q. This happens while Tcry is less than the Tc. At the end (after the while has ended) I calculate the gap of the SA, its execution time and I return the whole solution. This solution is the best one found by the SA, which means that it is either the same solution as the GHA (SA could not fine a better one) or a better solution with shorter makespan.

- sol_calc: The arguments of this function are the setups, loom sequence, assignments, number of machines and number of orders (jobs). The purpose of this function is to calculate the solution.

For each machine it creates a table that contains a row for each job. In each row (job) the following are stored: the initial machine load, setup of the job, assignment of the job and the final machine load (simply the sum of the previous three). It returns the tables (one for each machine) and the maximum machine load, which is the makespan of the solution

```python
if __name__ == "__main__":
    columns = ['instance_number', 'instance', 'number_of_machines', 'number_of_orders', 'number_of_workers', 'alpha',
               'lower_bound', 'Tc', 'Tcry', 'q', 'IT', 'gha_gap', 'gha_obj', 'sa_Gap', 'sa_obj', 'sa_Time']
    with open('results.csv', 'w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(columns)
    input_data = open('N_10_20_30_40_50.json')
    data = json.load(input_data)
    # loop over all instances in json
    for instance_number in data.keys():
        # reading from instance json
        lower_bound = data[instance_number]['LB']
        number_of_orders = data[instance_number]['Number_of_jobs']
        number_of_machines = data[instance_number]['Number_of_machines']
        alpha = data[instance_number]['Alpha']
        process = np.array(list(data[instance_number]['ProcessTimes'].values())).T
        setups = {}
        for m in data[instance_number]['SetupTimes'].keys():
            setups[int(m)] = np.array(list(data[instance_number]['SetupTimes'][str(m)]))
        # loop for each worker
        for number_of_workers in [5]:
            if number_of_workers >= number_of_machines:
                instance = str(instance_number) + "-Work" + str(number_of_workers)
                input_sols = open('Final_solutions_10_20_30_40_50.json')
                sols = json.load(input_sols)
                initial_sols = sols[instance]
                print(initial_sols)
                lower_bound = initial_sols['LB']
                gha_obj = initial_sols['Total_Upper_Bound']
                gha_gap = initial_sols['Gap']
                loom_seq = sols[instance]['Loom_Sequence']
                assignments = np.array(list(sols[instance]['Assignments'].values())).T
                for Tc in [200, 500]:
                    for Tcry in [1, 20]:
                        for q in [0.90, 0.95]:
                            for IT in [50,100]:
                                for iterations in range(5):
                                    sa_sols = simulated_annealing(process, setups, loom_seq, assignments, Tcry,
                                                                                                                    Tc, q, IT, lower_bound, number_of_machines, number_of_orders)
                                    results = [instance_number, instance, number_of_machines, number_of_orders,
                                               number_of_workers, alpha,
                                               lower_bound, Tc, Tcry, q, IT, gha_gap, gha_obj, sa_sols['Gap'], sa_sols['Obj'],
                                               sa_sols['Time']]

                                    with open('results.csv', 'a', newline='') as file:
                                        writer = csv.writer(file)
                                        writer.writerow(results)
```

Python Code Snippet 3: The 'Main' of the program

The third python code snippet is the main function. The main function is used to run the whole program and obtain the results. The first step is reading the files N_10_20_30_40_50.json and Final_solutions_10_20_30_40_50.json. The first file contains the Benchmark instances that were used in the paper regarding the evaluation of the Greedy Heuristic Algorithm. The second file contains the solutions of these Benchmark instances that the Greedy Heuristic Algorithm provided. It is very important to read the columns correctly. For my experiments I use the instances with 5 workers and I assume that there are no resource constrains (workers $\geq$ machines). In this way workers are always available when they are needed for setups and the complexity of the problem is reduced. This is because the load of each machine can be calculated as the sum of every job's setup and assignment in this machine, while on the other hand idle time could exist between jobs. Then I perform the experiments. Each experiment is performed 5 times to reduce the stochasticity of this specific implementation of the Simulated Annealing as it randomly changes jobs. The values of Tc, Tcry, q and IT are chosen based on relevant bibliography and experiments. The solutions of the SA are calculated every time by calling the function simulated_annealing() and then the results are written in a csv file.

```python
df = pd.read_csv('results.csv')
new_column_names = ['instance_number', 'instance', 'number_of_machines', '
    number_of_orders','number_of_workers', 'alpha', 'lower_bound', 'Tc', 'Tcry', 'q',
     'IT', 'gha_gap', 'gha_obj', 'sa_gap', 'sa_obj','sa_time']
df.columns = new_column_names
df_improved = df[df['gha_obj'] != df['sa_obj']]
df_improved #print(df_improved)
```

Python Code Snippet 4: Print the results of improved solutions

The fourth, and last, python code snippet reads the results.csv as a dataframe, adds the column names and then prints only the rows with the instances/solutions that the SA improved. A table with these instances is provided below (some columns are not displayed for brevity):

Machines	Jobs	Workers	Alpha	GHA_Gap	GHA_Obj	SA_Gap	SA_Obj	SA_time
2	10	5	0	15.669	148.38	10.282	141.47	0.599
2	10	5	2	8.149	132.06	8.133	132.04	0.269
5	10	5	0	8.972	37.05	8.913	37.03	0.369
5	10	5	1	10.29	44.85	7.36	43.66	0.494
5	10	5	2	13.506	55.4	11.663	54.5	0.305
2	20	5	0	11.622	320.79	10.060	316.30	1.480
2	20	5	2	7.962	241.11	7.811	240.77	0.450
5	20	5	0	10.758	90.42	9.840	89.67	2.044
2	30	5	0	10.427	552.03	9.531	547.55	0.685
2	30	5	2	7.478	416.06	7.254	415.19	1.987
5	30	5	2	9.64	125.09	9.482	124.91	1.600
5	40	5	0	12.407	183.63	11.899	182.80	0.447
5	40	5	2	7.615	169.26	7.323	168.80	3.419
2	50	5	2	9.356	695.12	9.193	694.08	0.963
5	50	5	0	12.636	210.73	12.150	209.82	3.373
5	50	5	2	8.501	229.15	8.108	228.32	0.375

Table 3: Improved Solutions with Simulated Annealing

In this table one can see the instances that the SA improved the solution of the GHA. For example in the first case that there are 2 machines, 10 jobs (orders), 5 workers and a = 0 the solution is improved as the gap is reduced from 15.669 to 10.282. In the SA_time I included the time of the specific experiment that found the better solution (Of course as I described for each instance the experiment is performed 5 times, while I also test different combinations of Tc, Tcry, q and IT). Is is important to note that for each experiment the execution time is generally low but in order to run all the tested experiments in Google Colab approximately 45 minutes are needed. I will further explain the results in the next and final chapter.

5 Conclusion and Future Work

To begin with, the initial instances (without taking into account the Tc, Tcry, q and IT) are 30. This implementation of Simulated Annealing manages to improve the solution in 16 instances, which is in more than 50% of the cases. Of course, one must take into account the stochasticity of the results regarding the approach of this implementation. Furthermore, in some cases the solution is improved slightly. In these cases the gap of the GHA is so small that the SA is difficult to reduce it way more (example: instance with 2 machines, 30 jobs, 5 workers, alpha = 2 with GHA gap = 7.478 and SA gap = 7.254). Generally, the GHA already provides great solutions with very small gap in the cases where the are no resource constraints. Last but not least, this implementation of SA changes jobs randomly without taking into consideration other important factors when generating perturbative solutions. One can think many different strategies that can be tested in this problem. For example, find the machine with the maximum load (makespan) and the job with the maximum setup time and swap it with the previous job or find the machine with the maximum load (makespan) and split the last job performed into other machines. These strategies may further improve the solutions but make the implementation of the SA a way more complex problem.

The next steps of this assignment is the testing of different strategies regarding the generation of perturbative solutions and the addition of the resource constraints in the implementation of the SA. In fact the indicative function for workers availability is presented in the Appendix. These next steps along with the creation of a genetic algorithm for tackling the problem will be the topic of my thesis project, which may lead to a publication.

6 Appendix

Below is the function regarding the resource constraints:

```python
def workers_availability(work, machines_time, setup_time):
    machines_time += 0.01
    work.append(machines_time)
    # print ("Machines time",machines_time)
    work.sort()
    # print(work)
    # print ("Workers list", workers_list)
    position_of_time = work.index(machines_time)
    # print(machines_time, setup_time)
    # print(work[position_of_time + 1])
    if position_of_time % 2 == 0:
        if machines_time + setup_time < work[position_of_time + 1]:
            machines_best = machines_time
        else:
            found = False
            while position_of_time + 3 < len(work) and not found:
                if work[position_of_time + 2] + setup_time < work[position_of_time +
    3]:
                    machines_best = work[position_of_time + 2] + 0.01
                    found = True
                else:
                    position_of_time += 2
    else:
        found = False
        while position_of_time + 2 < len(work) and not found:
            if work[position_of_time + 1] + setup_time < work[position_of_time + 2]:
                machines_best = work[position_of_time + 1] + 0.01
                found = True
            else:
                position_of_time += 2
    # print ("Machines best:",machines_best)
    # print ("Work",work)
    # print ("Machines best", machines_best - 0.01)
    return machines_best - 0.01
```

Python Code Snippet 5: Function for Resource Constraints