MAST90098
Approximation Algorithms and Heuristics
Project - 2016

Ria Szeredi, Kenneth Young, Lotte Romijn

October 16, 2016

Makespan Scheduling Problem (MS)

Schedule n jobs with designated processing times on m identical machines in such a way that the whole processing time is minimised.

Input:
Positive integers $p_1, p_2, ..., p_n$ and an integer $m \geq 2$ for some $n \in \mathbb{N} - \{0\}$. Each p_i is the processing time of the ith job on any of the m available machines.

Constraints:
For every input instance $(p_1, ..., p_n, m)$ of MS,
$\mathcal{M}(p_1, ..., p_n, m) = \{S_1, S_2, ..., S_m \mid S_i \subseteq \{1, 2, ..., n\}$ for $i = 1, ..., m$, $\cup_{k=1}^{m} S_k = \{1, 2, ..., n\}$, and
$S_i \cap S_j = \emptyset$ for $i \neq j\}$.

Cost:
For each $(S_1, ..., S_m) \in \mathcal{M}(p_1, ..., p_n, m)$,
$\text{cost}((S_1, ..., S_m), (p_1, ..., p_n, m)) = \max\{\sum_{l \in S_i} p_l | i = 1, ..., m\}$.

Goal:
minimum

1 Greedy Local Search Algorithm for MS

We implement a greedy local search (GLS) algorithm for MS, which picks the lowest cost neighbour at any iteration.

1.a. k-jump-neighbourhood

We consider a *k-jump*-neighbourhood:

We define a mapping $f_X : \mathcal{M}(x) \to 2^{\mathcal{M}(x)}$, such that if $\beta = \{\beta_1, \beta_2, ..., \beta_m\} \in f_x(\alpha)$ for some $\alpha = \{\alpha_1, \alpha_2, ..., \alpha_m\} \in \mathcal{M}(x)$ if β can be obtained from α by k "jumps".

Each jump allows some job $p \in S_i$ for some $i \in \{1, ..., m\}$ to move to any one of the m machines, including to the one in which it is contained. Hence, after one jump we have one of two cases:

- $\alpha_i \neq \beta_i$ and $\alpha_j \neq \beta_j$ for some $i \in \{1, ..., m\}$ and $j \in \{1, ..., m\}$, and $\alpha_\ell = \beta_\ell$ for $\ell \neq i$ and $\ell \neq j$, if job p jumps from machine i to machine j; or
- $\alpha = \beta$ if the job jumps to its own machine.

We thus define a "k-jump" as a sequence of k successive jumps.

We can check that the conditions for a suitable neighbourhood are satisfied. Let $\alpha = \{\alpha_1, ..., \alpha_m\}$ be an initial, feasible solution.

1. (*Reflexivity*): Since we allow a job to jump to the machine in which it is already contained, it is possible that, after k jumps, all jobs are in their original machines, and the solution is identical to α. Hence, we have that $\alpha \in f_X(\alpha)$ for every $\alpha \in \mathcal{M}(x)$. (We call this a 'loop').
2. (*Symmetry*): If $\beta \in f_X(\alpha)$ for some $\alpha \in \mathcal{M}(x)$, then $\alpha \in f_X(\beta)$, since reversing the k jumps from which β was obtained from α is also a k-jump.
3. (*Connectivity*): If we have any two feasible solutions α and β, we can always obtain one from the other by performing a finite number of k jumps (since we allow a job to jump to any machine). If $k \geq n$, one k-jump suffices, since we allow loops. If $k \leq n$, we perform maximally $\lceil \frac{n}{k} \rceil$ k-jumps. Hence, for all $\alpha, \beta \in \mathcal{M}(x)$, there exists a positive integer k and $\gamma_1, ..., \gamma_k \in \mathcal{M}(x)$ such that $\gamma_1 \in f_X(\alpha)$, $\gamma_{i+1} \in f_X(\gamma_i)$ for $i = 1, ..., k-1$, and $\beta \in f_X(\gamma_k)$.

The cardinality of this neighbourhood is quite large. If $k = 1$, we can move from one feasible solution of MS to another by moving one job, to the same or a different machine. If we have n jobs and m machines, there are $n\,m$ possible 1-jumps. For general k, there are $(n\,m)^k$ possible k-jumps.

1.b Implementation

We implement the *greedy local search* algorithm in Python (See *GLS* in MS_heuristics.py). We will call the resulting total processing time, produced by the algorithm, the 'makespan value'.

In our code for GLS, Variable-Depth Search (Section 2) and Simulated Annealing (Section 3), we represent a feasible solution $\{S_1, S_2, ..., S_m\}$ to the Makespan Scheduling problem as a list $\xi = [\xi_1, \xi_2, ..., \xi_n]$ of length n, where each $\xi_i \in [1, ..., m]$ represents in which machine job i is contained. Methods have been included which convert the solutions between these two representations (*convertSol_toListOfMachines* and *convertSol_toSetsOfJobs*).

To find the best *k-jump*-neighbour of a solution, we define a function *findBestNeighbour* which returns the best solution that can be reached from the input solution in one k-jump. It does so by systematically iterating through all possible combinations of jobs to jump and machines to jump to, and updating the solution if a better neighbour is found.

The GLS algorithm requires an initial, feasible solution for each instance. We consider two methods do generate initial solutions:

1. Random: assigning each job to a random machine (see *findInitialFeasibleSolution_rand*).
2. GMS - Greedy Makespan Scheduling algorithm (see *findInitialFeasibleSolution_GMS*)

The GMS algorithm works by first sorting the jobs by non-increasing processing time, assigning the m largest jobs each to a different machine, and assigning the remaining jobs to the least loaded machine. I.e., using the original problem notation:

1. We reindex the jobs such that $p_1 \geq p_2 \geq ... \geq p_n$
2. Let $S_i = \{\}$ for $i = 1, ..., m$
3. **For** $i = 1, ..., m$
   (a) $S_i := \{i\}$
   end for
4. **For** $j = m + 1, ..., n$ (only if $n > m$)

(a) $M := \min\{\sum_{l \in S_i} p_l | i = 1, ..., m\}$

(b) $S_M = S_M \cup \{j\}$

end for

5. **output**$(\{S_1, S_2, ..., S_m\})$

It can be shown that GMS is a 2-approximation algorithm for MS (Section 4.2.1 of the textbook).

We then convert the resulting initial solution $\{S_1, S_2, ..., S_m\}$ to the representation $\xi = [\xi_1, \xi_2, ..., \xi_n]$ of length n, where each $\xi_i \in \{1, ..., m\}$ represents in which machine job i is contained, by using *convertSol_toListOfMachines*, as described before. Both the random and GMS initial solutions are feasible solutions, since all jobs are assigned to one machine S_i, such that $\{S_1, S_2, ..., S_m \,|\, S_i \subseteq \{1, 2, ..., n\}$ for $i = 1, ..., m$, $\cup_{k=1}^{m} S_k = \{1, 2, ..., n\}$, and $S_i \cap S_j = \emptyset$ for $i \neq j\}$.

1.c Selecting the value of k

We perform an experimental study on randomly generated instances of MS across a range of n and m values in order to select an appropriate value for k.

We consider $n = 15, 20, 25, 30, 35, 40, 45, 50$, $m = 2, 4, 6, 8, 10$, and $k = 1, 2, 3$. For each combination of n, m and k, we generate 1 instance of MS. For each instance, we randomly select the processing times p_i, for $i = 1, ..., n$, uniformly distributed between 1 and 100. We determine the run time of our GLS implementation for each instance, and the solution quality by comparing the makespan value to a lower bound. We define this lower bound to be the least possible makespan value if we were able to process fractions of the jobs on any machine, i.e.

$$\text{lower bound} = \frac{\sum_{i=1}^{n} p_i}{m}. \tag{1}$$

We subsequently calculate the solution quality, denoting the makespan value produced by GLS as 'makespan',

$$\text{'makespan gap'} = \left(\frac{\text{makespan}}{\text{lower bound}} - 1 \right) \times 100\%, \tag{2}$$

which represents the percentage that the makespan value is above its lower bound.

The results of 1 realisation per n, m, and k, using a random initial solution, are given in Table 2 and Figure 1. We show three-dimensional plots of n versus m versus the makespan gap (in percentage) in Figures 1a, 1c, and 1e, and n versus m versus the run time (in seconds) in Figures 1b, 1d, and 1f. For

clarity, we only plot values for the makespan between 0 and 20 %, and values for the run time between 0 and 1200 seconds. We consider a 'reasonable' run time to be approximately 20 minutes (1200 seconds).

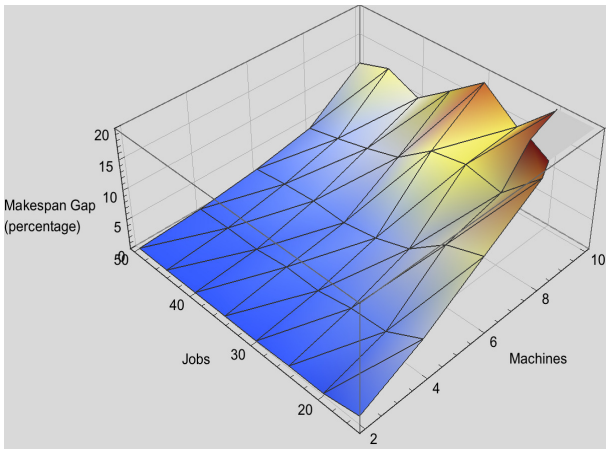
We can see in Tables 2a, 2c, and 2e that in general, the makespan gap decreases for each instance as we increase k. The average makespan gap over all n and m values is 6.98% for $k = 1$, 3.68% for $k = 2$, and 1.32% for $k = 3$. Thus, the average makespan gap improves as k increases. For all values of k, the makespan gap is larger for small n and large m, i.e. when few jobs need to be distributed over many machines. It is especially larger when n is close to m. This suggests that our lower bound is not such a great measure of solution quality for these instances. In fact, we know that for the case when $n = m = 10$ that the GLS algorithm returns the solution assigning one job to each machine, which can not be improved and thus must be optimal. For any instance when n is close to m, the lower bound provides a gross under-estimation of the true optimal; these are the cases when the possibility of each machine having similar total processing time is least likely. This shortcoming lead us to consider other methods of determining solution quality in Section 3.6.

In Table 2b, 2d, and 2f, we see that for all n and m, the run time increases as k increases. It increases rapidly: we see that the average run time over all combinations of n and m is 0.072 s for $k = 1$, 19.79 s for $k = 2$, and 1199 s for $k = 3$. As we increase k, the size of the neighbourhood increases by a factor of $n\,m$. However, for all combinations of n and m, the run time increases by a factor less than $n\,m$ as we increase k. Moreover, when k is larger and we increase n and m, the neighbourhood increases more (since the size is $(n\,m)^k$). Hence, in the tables we can see that the run time increases faster for larger k, as we increase n and m. In Figures 1b, 1d, and 1f, the run time has been plotted for $k = 1$, $k = 2$, $k = 3$, respectively. We have used the same range of the run time values (between 0 and 1200 s) for all three plots. Compared to $k = 3$, we can see that the run times for $k = 1$ and $k = 2$ are significantly smaller. However, for $n = 50$ and $m = 10$, the run time for $k = 2$ is approximately 3 minutes (195 seconds), and approximately 500 ($n\,m = 50 \times 10 = 500$) times larger than the run time for $k = 1$, which is 0.385 s.

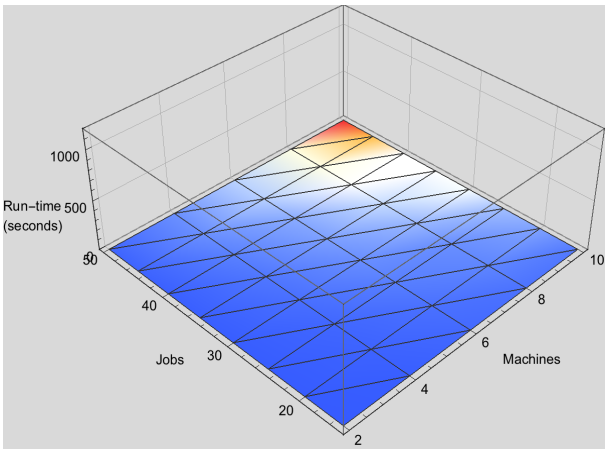
For our remaining experiments, we choose $k = 2$. In Figure 1d, we see that for all instances we have considered so far, the run time is smaller than 20 minutes (1200 seconds). Moreover, the makespan gap is smaller than for $k = 1$, for all instances except for $n = 15$ and $m = 8, 10$, i.e. when the number of jobs is relatively close to the number of machines. Hence, we make a trade-off between solution quality and run time, and select $k = 2$ as our k-value.

GLS Experiments

$k = 1$

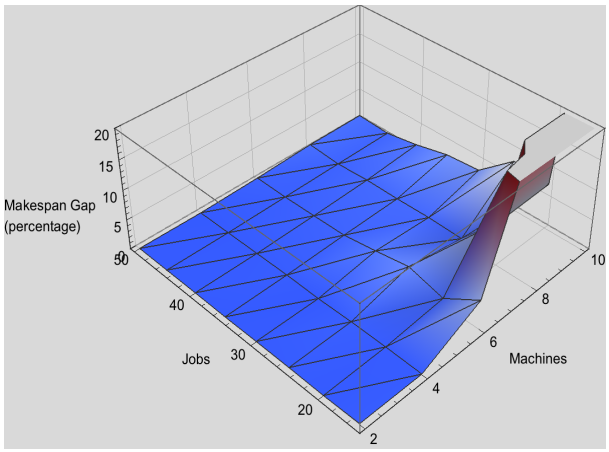


(a)

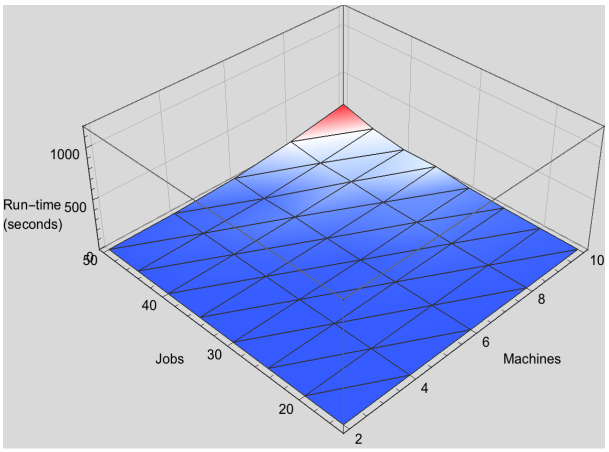


(b)

$k = 2$

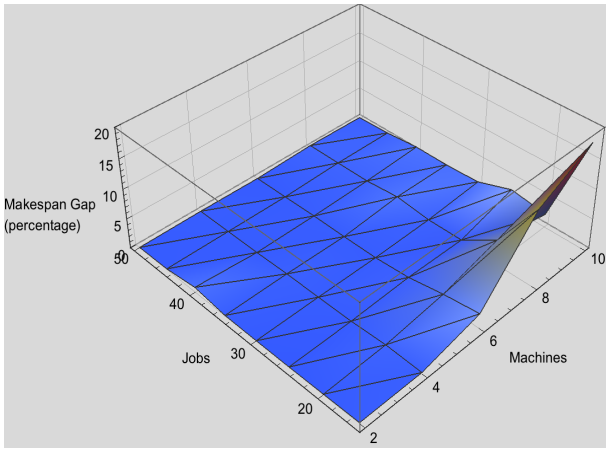


(c)

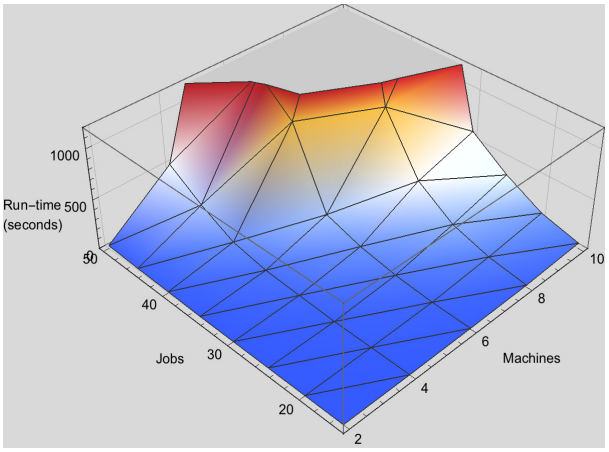


(d)

$k = 3$



(e)



(f)

FIGURE 1: **GLS Experiments: $k = 1, 2, 3$, using a random initial solution.** (a), (c) and (e) show the makespan gap (in percentage), and (b), (d), and (e) show the run time (in seconds).

1.d Experimental Study

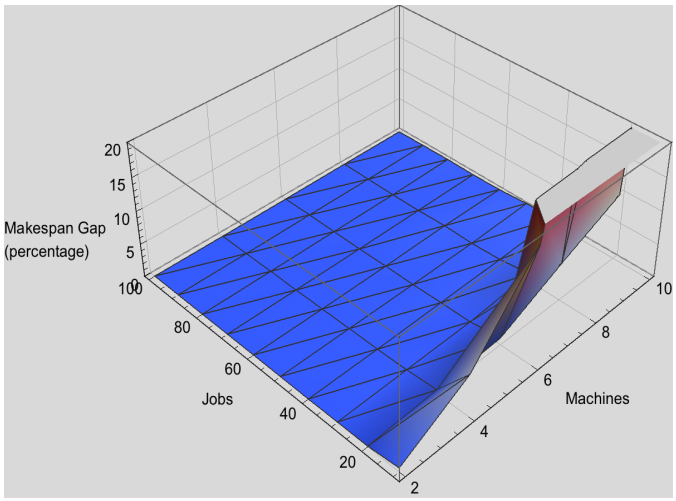
We perform an experimental study to test the performance of the GLS algorithm, using $k = 2$. We test for $n = 10, 20, 30, 40, 50, 60, 70, 80, 90, 100$ and $m = 2, 4, 6, 8, 10$. We also now test both methods (random and GMS) for finding an initial feasible solution.

For each combination of n and m, we generate 10 instances of MS. For each instance, we randomly select the processing times p_i, for $i = 1, ..., n$, uniformly distributed between 1 and 100. Each instance is generated with a different random seed.

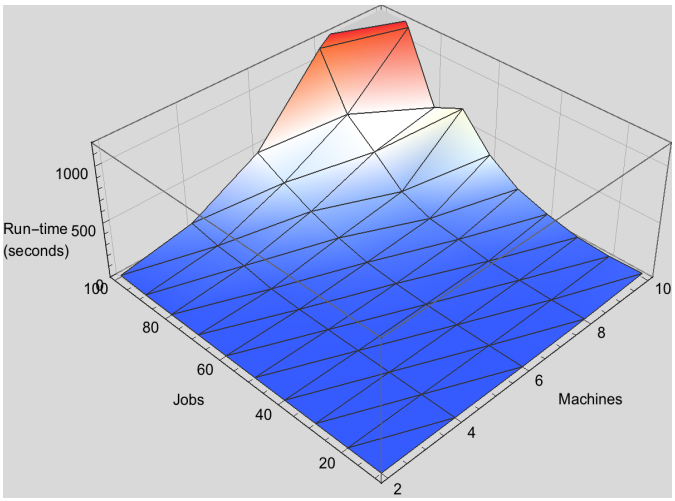
We consider a 'reasonable' run time to be approximately 20 minutes. The averaged results of 10 realisations per n, m, and k are presented in Table 3 and Figure 2. In Figure 2, only makespan gap values between 0 and 20 % are plotted, and run time values between 0 and 1200 s (20 minutes).

From Table 3a, we calculate that the average makespan gap over all n and m for the random initial solution is 4.09 %. In Table 3c, the average makespan gap over all n and m for the GMS initial solution is 4.11 %. In Table 3b, the average run time over all n and m for the random initial solution is 176.74 s. In Table 3d, the average run time over all n and m for the GMS initial solution is 171.04 s. Hence, the random initial solution and the GMS initial solution give similar results. This suggests that local minima can be found just as easily from either initial solution. We will also use these two methods for generating an initial solution when testing VDS and our heuristic.

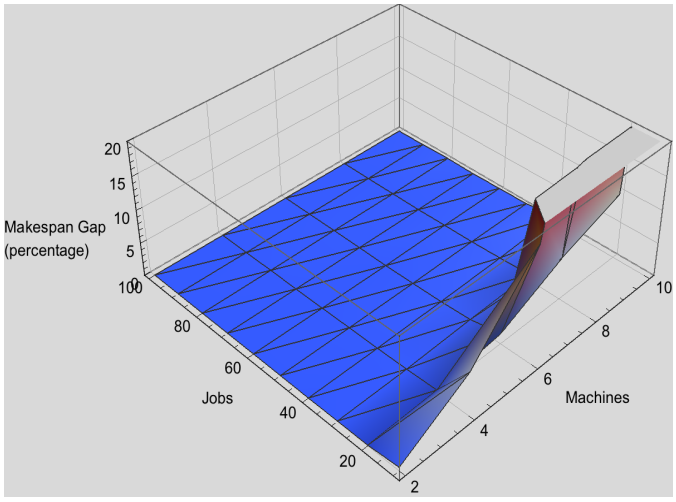
GLS experiments



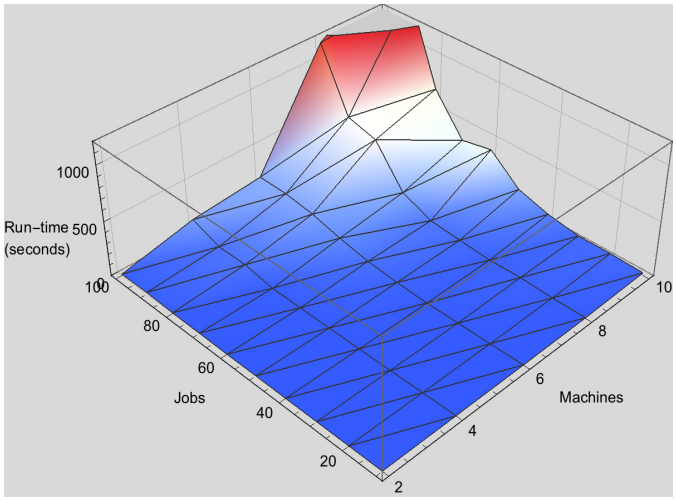
(a) Random



(b) Random



(c) GMS



(d) GMS

FIGURE 2: **Experiments GLS: k=2, using random and GMS initial solution.** (a) and (b) display the makespan gap (in percentage) and the run time (in seconds), using a random initial solution. (c) and (d) show the makespan gap (in percentage) and the run time (in seconds), using the GMS initial solution. We only plot makespan gap values between 0 and 20 %, and run time values between 0 and 1200 s.

2 Variable Depth Search Algorithm for MS

In this section, we consider the Variable-Depth Search (VDS) algorithm. This algorithm searches for an improved solution at a larger distance from the current solution, without performing an exhaustive search of the entire neigbourhood. Doing so gives us a higher chance of escaping local minima.

We first present detailed pseudo-code for the VDS, as well as provide a justification that our algorithm satisfies the required conditions. Next, we present the results of an experimental study on the same instances as those used in Section 1(d), and compare the results of the VDS to those of the GLS.

VDS 2.a

Pseudo-code for VDS

We begin with an input instance $I = (p_1, p_2, ..., p_n, m)$ of the Makespan Scheduling problem. Recall that, during implementation, we use a list $\xi = [\xi_1, \xi_2, ..., \xi_n]$ of length n, to represent a solution.

1. Generate an initial feasible solution $\xi = [\xi_1, \xi_2, ..., \xi_n]$
   (random or GMS described in Section 1)
2. IMPROVEMENT := TRUE

   EXCHANGE := $\{1, 2, ..., n\}$

   $J = 0$, $\alpha_J := \xi$
3. **while** IMPROVEMENT = TRUE **do**
   (a) **while** EXCHANGE $\neq \emptyset$ **do**
      i. $J := J + 1$
      ii. $\alpha_J := findBestNeighbour(I, \alpha_{J-1}, k,$ True, EXCHANGE)
      iii. EXCHANGE := EXCHANGE $-$ {jobs whose machines changed in the transition from α_{J-1} to α_J}

   end while

   (b) Compute $gain(\xi, \alpha_i) = \text{makespan}(\xi) - \text{makespan}(\alpha_i)$, for $i = 1, ..., J$

   (c) Compute $l \in \{1, ..., J\}$ such that
   $gain(\xi, \alpha_l) = \max\{gain(\xi, \alpha_i) \,|i \in \{1, 2, ..., J\}\}$

   (d) **if** $gain(\xi, \alpha_l) > 0$ **then**

      $\xi := \alpha_l$

      EXCHANGE := $\{1, 2, ..., n\}$

$J = 0$, $\alpha_J := \xi$

else

IMPROVEMENT := FALSE

end while

4. **output**(ξ)

Neighbourhood function and satisfaction of conditions in Section 3.6.2 of the textbook
Our neighbourhood function, *findBestNeighbour* (step (3)(a)(ii)), is the same function we used for GLS, but taking a few different input parameters which we will describe below. Finding the best neighbour of a feasible solution, α_i is equivalent to finding a feasible solution, α_{i+1}, which is such that $gain(\alpha_i, \alpha_{i+1}) = \max\{gain(\alpha_i, \delta) | \delta \in Neigh(\alpha_i)\} = \max\{\text{cost}(\alpha_i) - \text{cost}(\delta) | \delta \in Neigh(\alpha_i)\}$, and we only update α_i to α_{i+1} if $gain(\alpha_i, \alpha_{i+1}) > 0$. Thus condition (ii)(a) in Section 3.6.2 of the textbook is satisfied.

In order to satisfy the conditions (i) and (ii)(b), we have included two special input arguments to the *findBestNeighbour* method:

1. *differentSolRequired* is a Boolean parameter, which indicates whether we require a neighbour which differs by at least one ξ_i to the original solution. This is set to be True for the VDS algorithm (whereas for the GLS algorithm, we set it to False). The function does not update the best neighbour if the neighbour is the same as the input solution. The implementation of this parameter ensures that the final feasible solution has all jobs assigned to a different machine to the original solution. Thus condition (i) is satisfied.
2. *jobsToConsider* is a list representing the jobs whose assigned machine we are allowed to change. The *findBestNeighbour* method only iterates over the jobs in this list in its search for the best neighbour, thus returning a solution which is such that any jobs not in the list stay on their original machine. In the VDS algorithm, the input argument is the EXCHANGE list. Thus, at each iteration within the while loop in step 3.a, we obtain a neighbour which differs from the previous solution in the parameters of EXCHANGE only. This list is initialised as the set of all jobs, and is updated at each iteration of the while loop in step 3.a.iii by removing any jobs that were changed. This procedure is continued until there remain no jobs to be changed, after which we break out of the while loop in step 3.a, and condition (ii)(b) is thus satisfied.

Note that our implementation of the VDS algorithm differs slightly from the textbook. We reset J and α_J in step 3.d, displayed in red, which is not done in the textbook. In subsequent iterations of the while loop (step 3), we do not consider 'old' neighbours, i.e. neighbours found by previous iterations of the

inner while loop (step 3.a). We choose to do this because these old neighbours can not be better than the current best solution ξ, since otherwise we would have moved it before. And by this procedure, we only need to check the value of J gains every iteration of the while loop (step 3), and we terminate if we can not find a better solution among any of the J neighbours generated.

VDS 2.b

We use Python to implement the *Variable Depth Search*. See *VDS* in MS_heuristics.py.

VDS 2.c

Experimental results

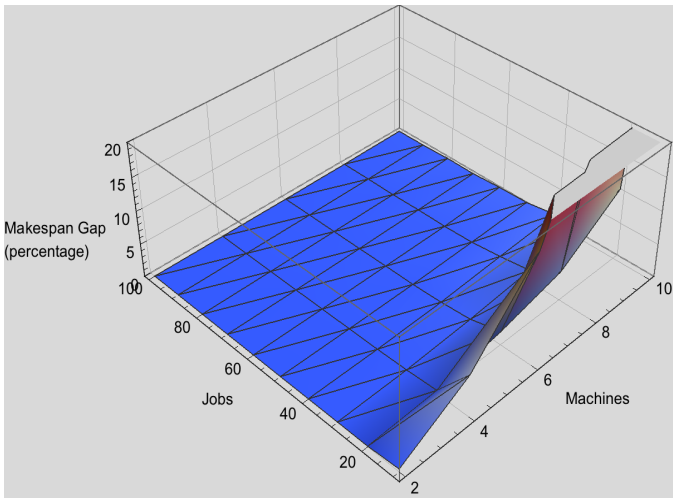
We perform an experimental study for VDS using the same instances that were generated for Section 1(d), where we generated 10 instances for each combination of n and m. The averaged results are displayed in Table 4 and Figure 3.

From Table 4a, we calculate that the average makespan gap over all n and m for the random initial solution is 3.38 %. In Table 4c, the average makespan gap over all n and m for the GMS initial solution is 3.28 %. In Table 4b, the average run time over all n and m for the random initial solution is 79.30 s. In Table 4d, the average run time over all n and m for the GMS initial solution is 42.69 s. Hence, a local minimum is reached faster from the GMS initial solution than from the random initial solution, but the values of the reached local minima are similar for both initial solution methods. Hence, a best solution is found more easily from a GMS initial solution, which has a makespan value which is less than twice the optimal makespan value, whereas in a random initial solution the initial makespan value can be as high as the sum of all processing times (if all jobs are put in the same machine).

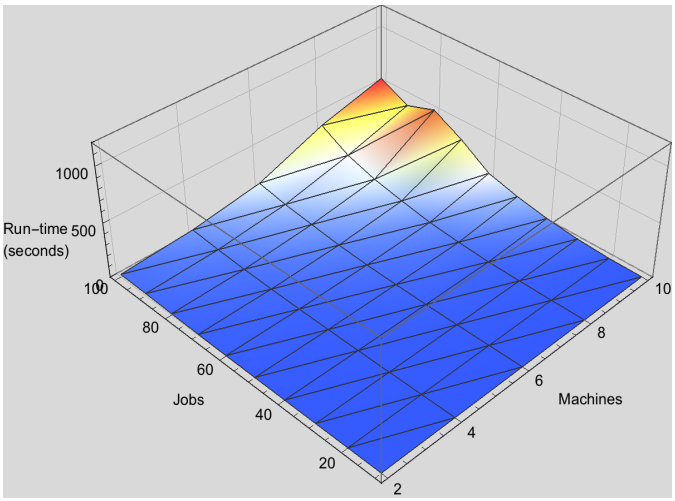
Comparison to GLS

By comparing Table 3 with Table 4, we can see that both the makespan gap and the run time of VDS are generally lower than the makespan gap and the run time of GLS, for all combinations of n and m. The averages described above are also lower. This is expected since the VDS algorithm performs several local k-jumps in order to see whether there are better solutions at larger distances from the current best solution, whereas the GLS algorithm only iterates to the nearest local minimum from the initial feasible solution. Since VDS is better than GLS, there are multiple local minima in our neighbourhood.

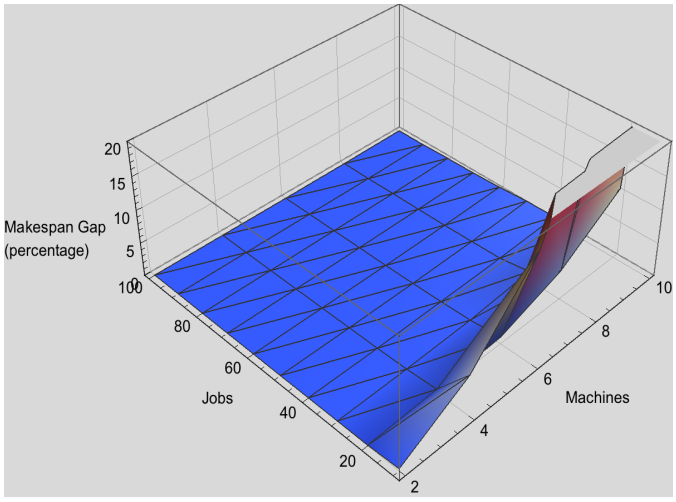
VDS Experiments



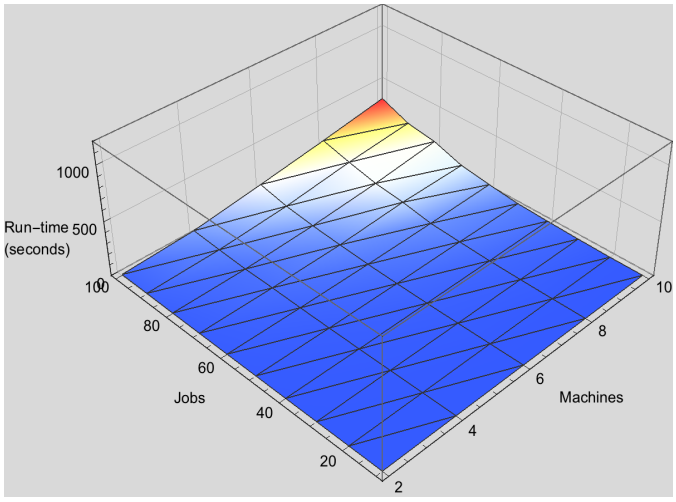
(a) Random



(b) Random



(c) GMS



(d) GMS

FIGURE 3: **Experiments VDS: k=2, using random and GMS initial solution.** (a) and (b) display the makespan gap (in percentage) and the run time (in seconds), using a random initial solution. (c) and (d) show the makespan gap (in percentage) and the run time (in seconds), using the GMS initial solution. We only plot makespan gap values between 0 and 20 %, and run time values between 0 and 1200 s.

3 Simulated Annealing Algorithm for MS

The heuristic we choose to implement is simulated annealing, for which we present detailed pseudo-code in Section 3.1. This algorithm has several free parameters, namely, the neighbourhood function, the cooling schedule which determines the rate at which the temperature decreases, and the initial value of the temperature. We use the same neighbourhood as defined in Section 1.

In Section 3.2, we consider an algorithm developed by Ben-Ameur [2004] to find a good initial value of the temperature. Several cooling schedules are considered, and the results of experiments performed on each are presented in Section 3.3. Section 3.4 contains the results of comparing the performance of the simulated annealing heuristic to the performance of the GLS and VDS algorithms. This then motivates Section 3.5, in which we push our heuristic to its limits to find the largest possible input instances which can be handled. Finally, in Section 3.6, we conclude by providing other possible methods of determining solution quality and how our heuristic performs with respect to each one.

3.1 Simulated Annealing Algorithm Pseudocode

In our algorithm, we make a slight modification to keep track of the best solution found by the algorithm (Step (3)). In the final step, the best solution is compared to the final solution found by the simulated annealing algorithm. We do this in order to avoid the algorithm returning a worse solution than one previously found.

We again use the random or GMS initial feasible solutions, described in Section 1.

Input: An input instance $I = (p_1, p_2, ..., p_n, m)$ of the Makespan Scheduling problem.

1. Generate an initial feasible solution $\xi = \{\xi_1, \xi_2, ..., \xi_n\}$, using Random or GMS.
   $T_0 := 1.5 \times \max\{\, p_i \,|i = 1, ..., n\}$ (See below for a discussion)
   $T := T_0$
2. $I := 0$
   $best := \xi$
3. **while** $T > 0$ **do**
   (a) Randomly select a neighbour $\beta \in Neigh_x(\alpha)$
   (b) **if** makespan(β) $\leq$ makespan(α) **then**
      i. $\alpha := \beta$
   else

i. Generate a random number r uniformly in the range $(0, 1)$

ii. **if** $r < e^{-\frac{\text{makespan}(\alpha)-\text{makespan}(\beta)}{T}}$ **then**

$\alpha := \beta$

(c) **if** makespan(α) < makespan($best$) **then**

$best := \alpha$

(d) $I := I + 1$

(e) $T :=$ reduceTemperature(T_0, I)

end while

4. **if** makespan(α) < makespan($best$)
output(α)
else
output($best$)

3.2 Choice of the Initial Temperature

The performance of the algorithm can vary greatly depending on the initial value of the temperature. We follow the work of Ben-Ameur [Ben-Ameur, 2004], who developed an algorithm to pre-determine the initial temperature T_0.

In the simulated annealing algorithm presented above, the probability of accepting a bad move (with cost increase) from solution α to β is

$$e^{-\frac{\text{cost}(\alpha)-\text{cost}(\beta)}{T}} = e^{-\frac{\text{makespan}(\alpha)-\text{makespan}(\beta)}{T}}, \tag{3}$$

where T is the current temperature, and $\text{cost}(\alpha) > \text{cost}(\beta)$. We consider the average probability of accepting a bad move as

$$\pi_\alpha = \frac{e^{-\frac{\text{cost}(\alpha)-\text{cost}(\beta)}{T}}}{\sum_j e^{-\frac{\text{cost}(\alpha)-\text{cost}(\beta_j)}{T}}}, \tag{4}$$

where we sum over all possible neighbours β_j of α (and assume the generation probability of reaching a neighbour is symmetrical, see Ben-Ameur [2004]). The idea of Ben-Ameur [2004] is to choose the initial temperature such that the acceptance probability is close to a certain value χ_0 by estimating this acceptance probability based on a random set of neighbours of α which increase the cost. For a full discussion of the algorithm, please refer to Ben-Ameur [2004]. The pseudo-code of our implementation is described below:

Algorithm 3.2: Initial Temperature Generation

Input:

- The input instance $I = (p_1, ..., p_n, m)$
- $k = 2$ exchange value
- The desired acceptance probability $\chi_0 = 0.8$
- The number of random transitions to generate $S = 1000$
- A stopping criterion $\epsilon = 0.001$

1. Generate a set, $\mathcal{S}$, of size S, of random transitions:

   For each random transition, we generate a random initial solution α and move to a random neighbour β. We let $E_{\max} := \min\{\text{cost}(\alpha), \text{cost}(\beta)\}$ and $E_{\min} := \max\{\text{cost}(\alpha), \text{cost}(\beta)\}$ for each transition S.

2. $T_1 := -\frac{\sum_{t \in S} E_{\max_t} - E_{\min_t}}{|S| \ln(\xi_0)}$

   Let $n = 1$

   Compute $\chi(T) := \dfrac{\sum_{t \in \mathcal{S}} e^{\frac{E_{max_t}}{T_n}}}{\sum_{t \in \mathcal{S}} e^{\frac{E_{min_t}}{T_n}}}$

3. **while** $|\chi(T_n) - \chi_0| > \epsilon$ **do**
   (a) $T_{n+1} := T_n \cdot (\frac{\ln(\chi(T_n))}{\ln(\chi_0)})^{\frac{1}{p}}$
   (b) Compute $\chi(T_n)$

4. **output** T_n

We will use this algorithm to generate the initial temperature for test cases with $n = 10$, 20, 30, 40, 50, 60, 70, 80, 90, 100 and $m = 2$, 4, 6, 8, 10, in order to get an idea of how large the initial temperature should be for these instances. We do this 100 times per combination of n and m, with processing times uniformly distributed between 1 and 100, and calculate the average. The averaged results are shown in Table 5 and Figure 4. We see that the initial temperature is larger when the number of machines is small, and the number of jobs is large. There is more variation in the initial temperature varying the number of machines than the number of jobs. By performing a few more test instances with smaller and larger processing times (with self-made instances and not shown in this report), we found that choosing an initial temperature of approximately 1.5 times larger than the largest processing time in an instance gives good results (and is faster, since we do not need to call the algorithm by Ben-Ameur [2004] for

every run of our simulated annealing algorithm). We can also see that this approximately holds for the instances in Figure 4, since the processing times are uniformly distributed between 1 and 100, and the temperature is approximately 150 for a small number of machines.

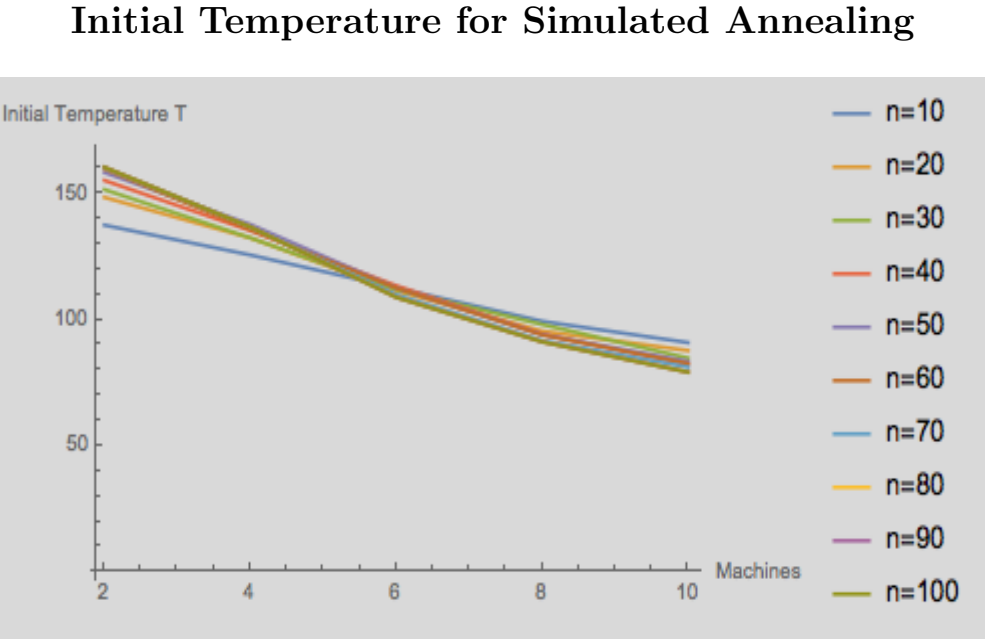


FIGURE 4: The initial temperatures generated by the algorithm by Ben-Ameur [2004], using 100 realisations per n and m, processing times between 1 and 100, and the random initial solution.

3.3 Choice of the Temperature Reduction Function: *reduceTemperature*

We consider several options for the cooling schedule, which is called $reduceTemperature(T_0, I)$ in the pseudocode of Section 3.1. All are functions, f, of the initial temperature, T_0, and the iteration number, I.

Exponential multiplicative cooling

$f_1(T_0, I) = T_0 \cdot \mu^I$

where we set $\mu = 0.8$.

Simple exponential cooling

$f_2(T_0, I) = T_0 - I$

Linear multiplicative cooling

$$f_3(T_0, I) = \frac{1}{1+I} \cdot T_0$$

Quadratic multiplicative cooling

$$f_4(T_0, I) = \frac{1}{1+I^2} \cdot T_0$$

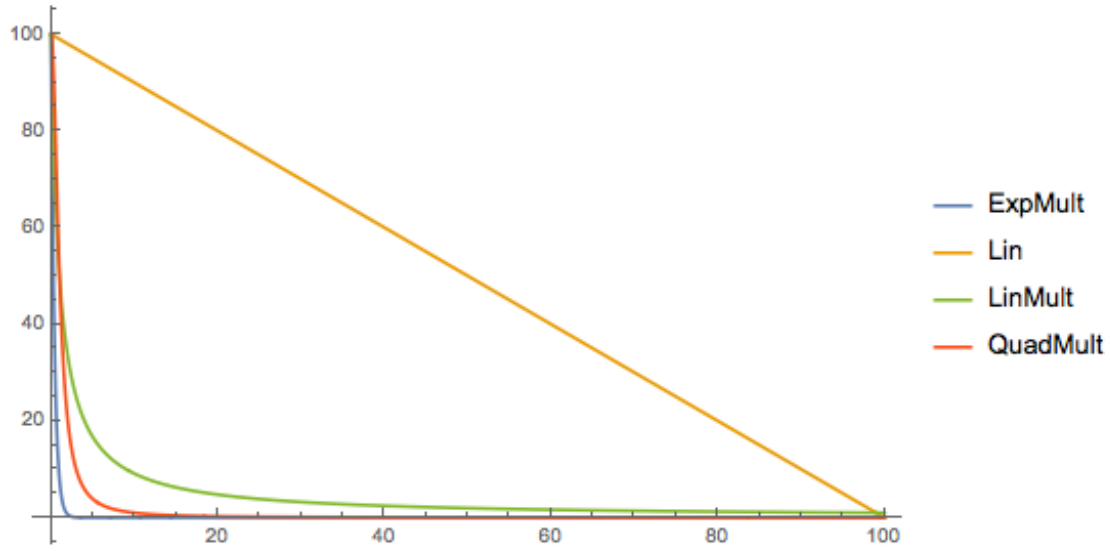


FIGURE 5: Cooling schedule plots for $T_0 = 100$.

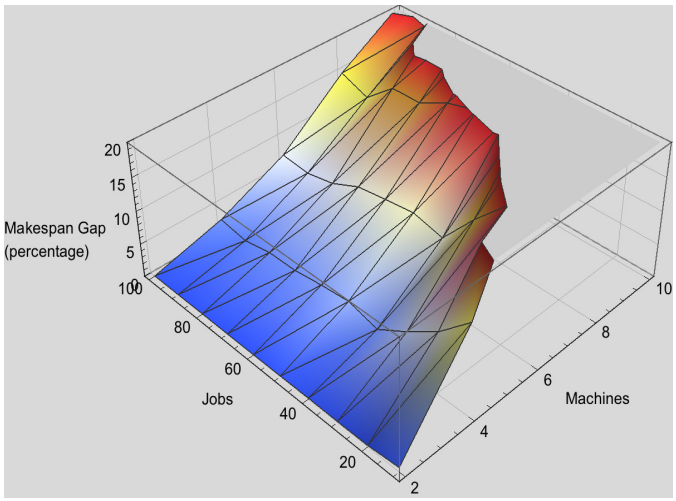
We experiment with these four cooling functions for the simulated annealing algorithm. We implemented the simulated annealing algorithm of Section 3.1 in Python. See *ourHeuristic* in MS_heuristics.py.

For $n = 10, 20, 30, 40, 50, 60, 70, 80, 90, 100$ and $m = 2, 4, 6, 8, 10$, we run our simulated annealing algorithm 10 times, using a random initial solution. The averaged results are given in Table 6 and 7, and in Figure 6 and 7.

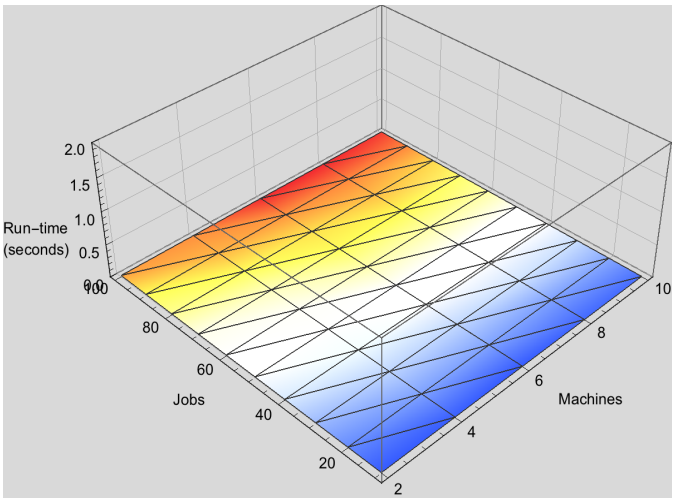
We find that the quadratic multiplicative function is by far the slowest, taking between ten and a hundred times longer than the other methods. However, the makespan gap is always lower, and more than 10 times lower when $n > 50$ and $m > 6$ for the quadratic multiplicative method. We see that when the number of jobs is close to the number of machines, the makespan gaps are comparable for all methods. Because the quadratic multiplicative method has a better solution quality, we choose to continue with this cooling schedule.

In the next section, we investigate the best choice of initial solution to use.

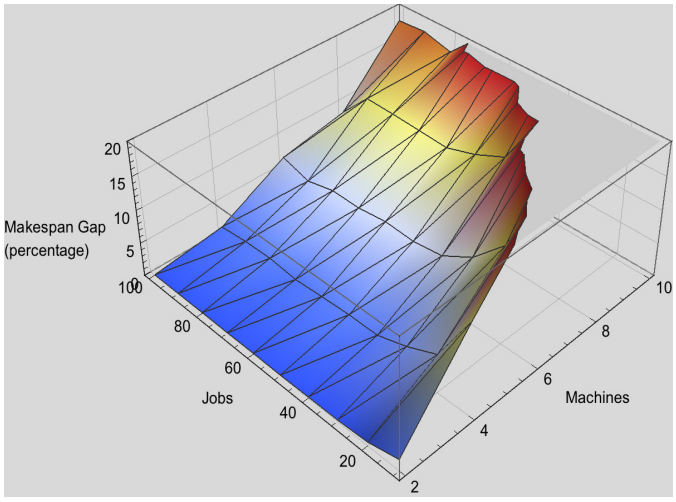
Simulated Annealing Cooling Function Experiments



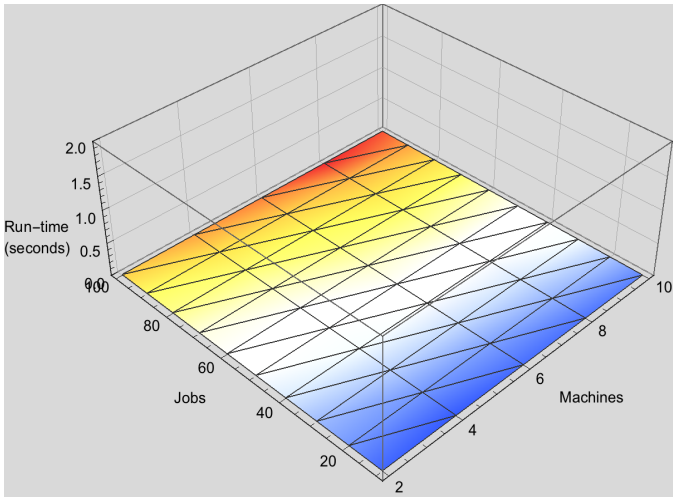
(a) Simple Linear Makespan Gap



(b) Simple Linear Run time



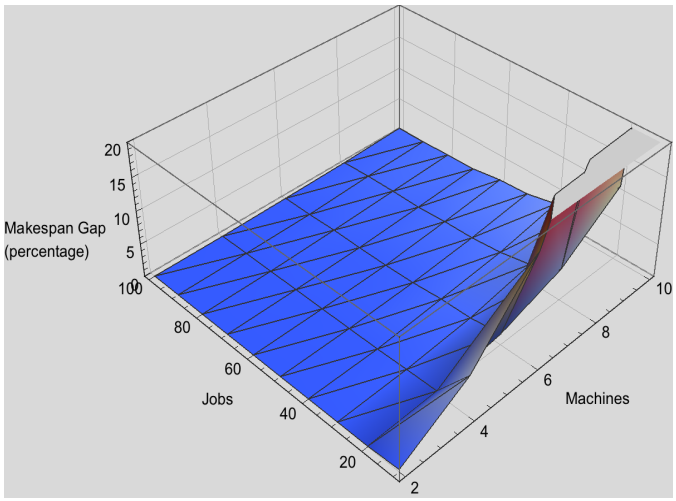
(c) Linear Multiplicative Makespan Gap



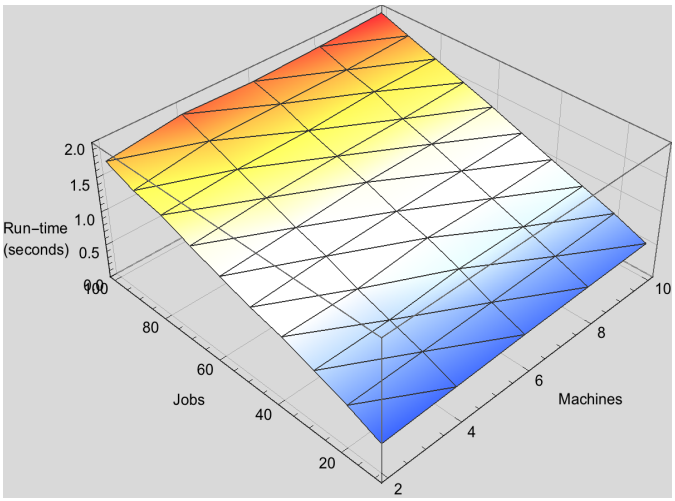
(d) Linear Multiplicative Run time

FIGURE 6: **Experiments Simulated Annealing: simple linear and linear multiplcative cooling functions.** (a) and (b) display the makespan gap (in percentage) and the run time (in seconds), using a simple linear cooling function. (c) and (d) show the makespan gap (in percentage) and the run time (in seconds), using a linear multiplicative cooling function. We only plot makespan gap values between 0 and 20 %, and run time values between 0 and 1200 s.

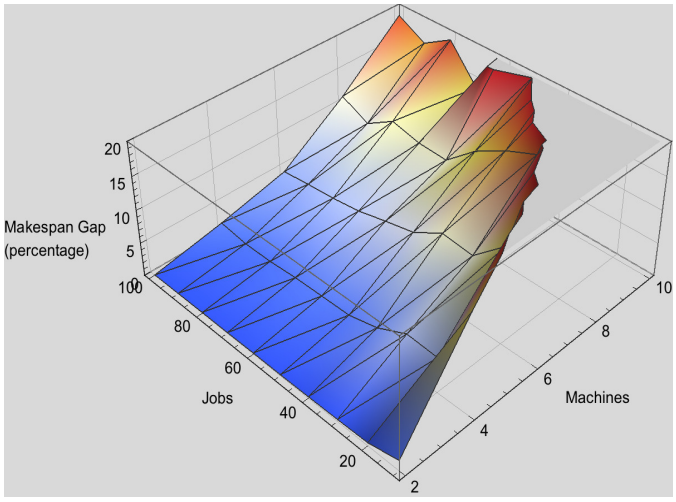
Simulated Annealing Cooling Function Experiments



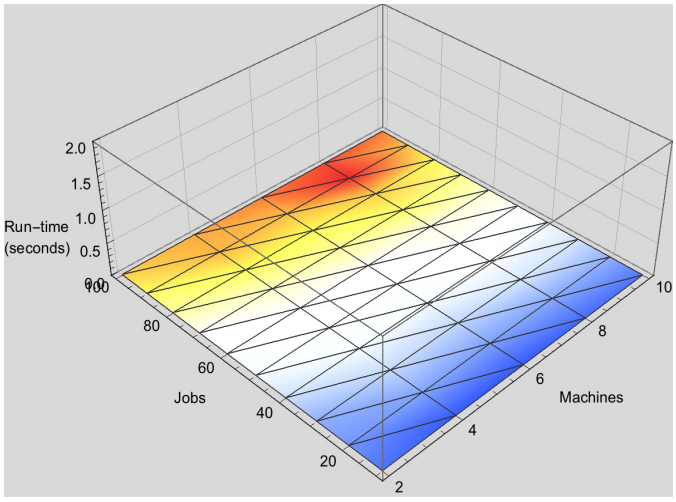
(a) Quadratic Multiplicative Makespan Gap



(b) Quadratic Multiplicative Runtime



(c) Exponential Multiplicative Makespan Gap



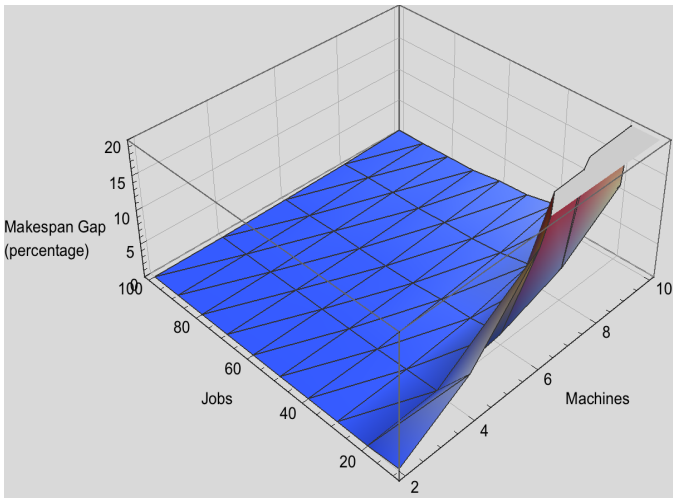
(d) Exponential Multiplicative Runtime

FIGURE 7: **Experiments Simulated Annealing: quadratic and exponential multiplicative cooling functions.** (a) and (b) display the makespan gap (in percentage) and the run time (in seconds), using a quadratic multiplicative cooling function. (c) and (d) show the makespan gap (in percentage) and the run time (in seconds), using a exponential multiplicative cooling function. We only plot makespan gap values between 0 and 20 %, and run time values between 0 and 1200 s.

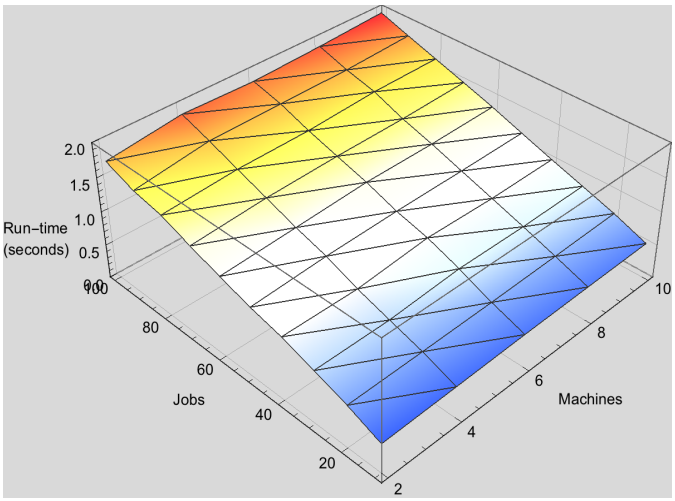
Choice of Initial Solution

We test both random and GMS initial solutions for our heuristic procedure. The results of 10 realisations per n and m are shown in Table 8 and Figure 8. We see that the random and GMS initial solution give similar results. The average makespan gap over all n and m for the random initial solution is 3.69 %, while for the GMS initial solution it is 3.55 %. The average run time over all n and m for the random initial solution is 1.19 s, while for the GMS initial solution it is 1.22 s. Hence, the performance of our simulated annealing algorithm is not affected by a choice between these two initial solution methods. Note that the run time does not depend much on the number of machines m, but on the number of jobs n.

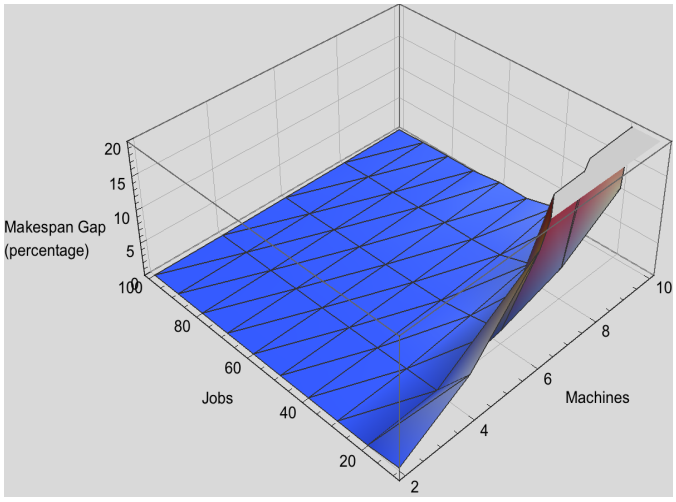
Simulated Annealing Initial Solution Experiments



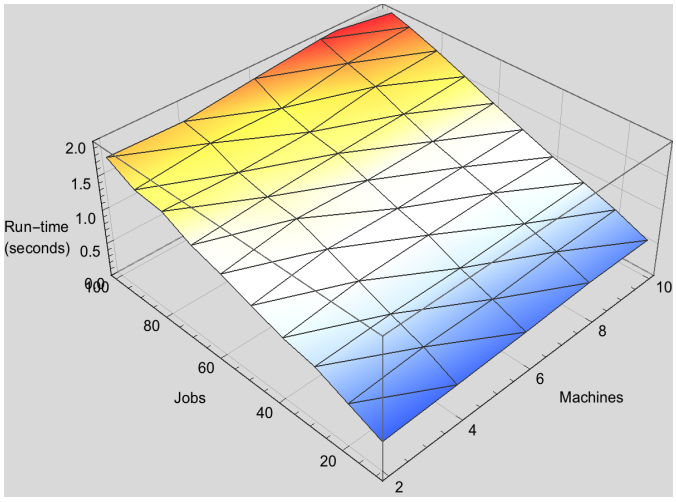
(a) Random



(b) Random



(c) GMS



(d) GMS

FIGURE 8: **Experiments Simulated Annealing: random and GMS initial solutions.** (a) and (b) display the makespan gap (in percentage) and the run time (in seconds), using a random initial solution. (c) and (d) show the makespan gap (in percentage) and the run time (in seconds), using the GMS initial solution. We only plot makespan gap values between 0 and 20 %, and run time values between 0 and 1200 s.

3.4 Comparison to GLS and VDS

We can compare the makespan gaps and run times of our simulated annealing solutions with GLS and VDS. Comparing tables 3, 4, and 8, and their respective figures 2, 3, and 8, we see that the makespan gaps by the simulated annealing algorithm are comparable to the makespan gaps by VDS. However, the run time of the simulated annealing algorithm is much faster, with average run time over all n and m (1.19 s for the random initial solution and 1.22 s for the GMS initial solution) much faster than the average run time of VDS (79.30 s for the random initial solution and 42.69 s for the GMS initial solution), and more than 100 times faster than GLS (176.74 s for the random initial solution and 171.04 s for the GMS initial solution).

In addition, in Table 4 and 8, we see that the makespan gap when n and m are comparable (or equal when $n = m = 10$) is approximately equal for VDS and simulated annealing. As noted in Section 1, our measure of solution quality (makespan gap) is bad in these cases. However, we can see that VDS and simulated annealing have similar results for these cases. In Section 3.6, we will address this issue and compare the results of our simulated annealing algorithm to solutions given by an exact algorithm (MIP).

3.5 Performance Limitations

To test the performance limitations of the simulated annealing algorithm we experiment with very large input instances. By choosing to use the slowest of the cooling functions (i.e. quadratic multiplicative cooling) we may have limited the size of input instances that our algorithm can handle in a reasonable amount of time. Even so, we found instances with 10,000 jobs still terminate within 10 minutes.

So far the input instances we generated had a uniform distribution of processing times between 1 and some constant maximum value. We chose the maximum processing time to be 100 for all the instances we generated for testing the limitations of the GLS and VDS algorithms. However, the processing times of the jobs and their distribution could also influence the solution quality.

In order to explore this, we let the maximum processing time be defined as a function of n, namely: max-processing-time $= n/10$. The division by 10 here is an artefact of our heuristic being implemented in Python, as when we tested the input instance of 1,000,000 jobs and 2 machines, calculating the total load of a machine required storing large scalar values. Without the assistance of 3rd party packages to handle memory allocation, Python cannot store scalar values larger than 2^{31} and so this calculation resulted in overflow errors. Simply dividing by 10 solved this issue for the size of the instances we aimed to test.

For these experiments we vary the number of jobs but keep the number of machines constant at $m = 2$. Our initial testing indicated that when n became 2 orders of magnitude larger than m, the exact number of machines began to have very little influence on the results. Note, in Table 8b and 8d and Figure 8, the run time does not vary much with m. So, instead of testing $n = 100$, 500, 1000, 5,000, 10,000 and 50,000 against a range of m values, we chose m to be a constant, thus saving computation time and still achieving the desired outcome of testing simulated annealing's limits.

3.5.1 Engineering Poor Input Instances

We aim to create instances of the problem which are particularly difficult to solve to optimality. To this end, we investigate ways to distribute the processing times between 1 and $n/10$ other than simply using the uniform distribution. We consider instances where the processing times were almost exclusively concentrated near the extreme values, i.e. the minimum and maximum. We introduce the following definitions before the formulation of the poor distribution is provided.

ϱ : Maximum possible processing time.

κ : The concentration of the processing times in the extremes.

Π : Probability of a job having a processing time from one extreme.

π : Probability of a job having a processing time not from an extreme.

N : Number of processing times assigned a large probability in each extreme, $N = \lfloor \kappa\varrho \rfloor$.

ν : Number of processing times assigned a small probability, $\nu = \varrho - 2N$.

Poor Distribution Definition

Let X be a random variable with probability density function, f, defined as follows,

$$P(X = x) = f(x) := \begin{cases} \Pi/N & 1 \leq x < \kappa\varrho, \\ \pi/\nu & \kappa\varrho \leq x < (1-\kappa)\varrho, \\ \Pi/N & (1-\kappa)\varrho \leq x \leq \varrho. \end{cases}$$

For the our experiments we chose $\Pi = 0.45$, $\pi = 0.1$ and $\kappa = 0.01$. So 90% of the total probability was concentrated in 2% of the possible processing times. To implement this probability density function, we select processing times using Python's (Numpy) random inbuilt random number generator, which takes the probability density function as input.

We are motivated to try such a distribution as the makespan values between most neighbouring solutions will have a high level of volatility. This should result from the distribution heavily weighting the

concentration of processing times to within a percent of 1 or ϱ. When the simulated annealing algorithm considers a random neighbour to move to, this neighbour will likely have an approximately equal makespan to our current solution or a drastically differing one. So the neighbourhood of any feasible solution will contain a large amount of exchanges that have little if any affect on the makespan (when moving small jobs), together with a large amount of neighbours which will have wildly large affects (when moving large jobs).

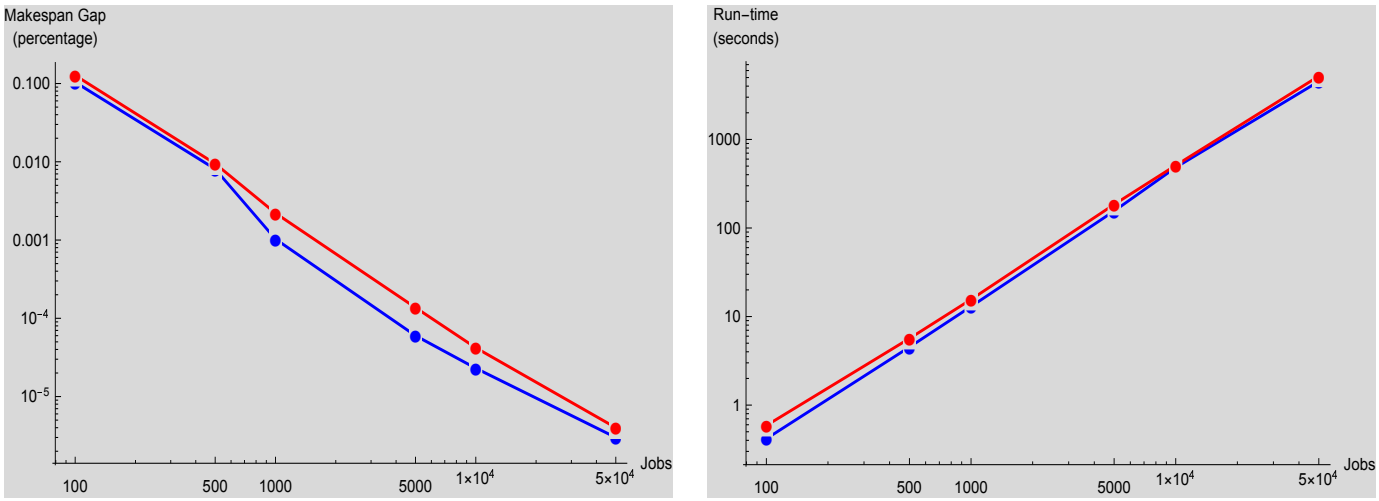
An instance with such a poor distribution of processing times will have a volatile neighbourhood topology. This should result in simulated annealing having more difficulty in finding the good local optima.

3.5.2 Results of Performance Limits Experiments

The averaged results of 10 realisations for large n and $m = 2$ are shown in Tables 9 and Figure 9, for both the uniform and the poor distributions of processing times between 0 and $n/10$. We see that the makespan gap decreases with approximate slope 1, as n increases. This gives that the makespan gap is of $O(1/n)$. The run time increases as n increases and is approximately of $O(n^2)$.

In Figure 9b, where we have plotted the run time results for the $n = 100$, 500, 1000, 5,000, 10,000 and 50,000, we can see that for the largest number of jobs considered our simulated annealing algorithm has taken longer than the specified 20 minutes to terminate. Specifically, the algorithm required on average 75 minutes for a uniform distribution of processing time and an average of 86 minutes for the poor distribution (see tables 9b and 9d). But for $n = 10,000$, our heuristic terminated within 10 minutes for both distributions tested. Noting that Figure 9b is a LogLog plot, we can see that the largest input instance our heuristic can handle in a reasonable amount of time has $n \approx$ 20,000-30,000.

We also find that when the instance size became exceedingly large, using a uniform distribution of processing times (in blue) made the simulated annealing algorithm find close to optimal solutions more easily than the new distribution (in red). This indicates that our poor distribution of processing times is slightly more difficult for simulated annealing to solve. Moreover, as n increases we can conclude that the large and small processing times in our pathological distribution 'even out' and make the problem easier to solve.



(a) Makespan Gap

(b) Run time

FIGURE 9: **Experiments Simulated Annealing.** Blue line: uniform distribution, red line: pathological distribution. (a) and (b) display the makespan gap (in percentage) and the run time (in seconds), using both types of distributions. We only plot makespan gap values between 0 and 20 %, and run time values between 0 and 1200 s. The results are also given in Table 9.

3.6 Methods of Determining Solution Quality

So far, we have measured the solution quality by calculating the makespan gap, as explained in Section 1, Equation (2). We saw that when n is close to m, the makespan gap is quite large, and also when $n = m = 10$ although the solution is optimal. The lower bound we calculated with Equation (1), assuming the processing times of the jobs are fractional, is too low because it is less likely that each machine has a similar total processing time.

3.6.1 Other measures of solution quality

Another possible strategy for measuring the solution quality is to calculate the ratio between of the makespan output by GMS, described in Section 1, and the makespan output of our simulated annealing algorithm. When we take the GMS as initial solution in the simulated annealing algorithm, the final makespan of our heuristic can not be worse than GMS, since we save the current best solution and output this if no better solution is found. We re-run the simulated annealing algorithm for the instances of MS we saved and used for GLS and VDS in Table 3 and 4, and also determine the optimal solution using .... GMS is an 2-approximation algorithm....

3.6.2 Constraint Programming

We can calculate whether our solution is optimal by taking the ratio between the solution output by our simulated annealing algorithm and the optimal solution.

Check approximation ratio GMS!!! It should be 2

Detail on true implementation of MP, in bash?

insert table with 100% in all entries here

Average Optimality Ratio

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.002	0	0	0	0	0	0	0	0	0
4	0.000	0.003	0.001	0.001	0.000	0.000	0.000	0.000	0.000	0
6	0	0.013	0.007	0.004	0.003	0.002	0.001	0.001	0.001	0.001
8	0	0.014	0.019	0.012	0.007	0.004	0.004	0.003	0.003	0.002
10	0	0.003	0.042	0.048	0.017	0.008	0.006	0.007	0.002	0.001

TABLE 1: The average ratio minus 1 of our simulated annealing solution to the optimal solution (by CP), for instances that were solved to optimality within 5 minutes.

4 Conclusion

We conclude that our simulated annealing algorithm is much faster than GLS and VDS for the MS input instances we have seen, while producing solutions of similar quality as VDS. We initially measured the quality of GLS, VDS, and our heuristic using the makespan gap, which uses the lower bound of the total processing time assuming all processing times are fractional.....

References

Ben-Ameur, W. (2004). Computing the initial temperature of simulated annealing. *Computational Optimization and Applications*, 29(3):369–385.

5 Appendix

Full results for GLS experiments

GLS Experiments with Random Initial Solution

$k = 1$

(a) Makespan gap

m \ n	15	20	25	30	35	40	45	50
2	1.40	0.35	0.34	0.15	0.14	0.09	0.04	0.07
4	5.93	2.84	2.51	1.43	1.56	0.84	0.88	0.69
6	11.99	10.06	5.62	3.73	2.74	3.00	2.58	1.93
8	17.85	12.18	13.44	12.42	8.11	6.45	5.03	3.81
10	40.01	19.70	14.06	18.40	14.12	9.98	12.30	10.50

(b) Run time

m \ n	15	20	25	30	35	40	45	50
2	0.001	0.002	0.003	0.003	0.005	0.005	0.008	0.009
4	0.004	0.008	0.011	0.018	0.024	0.032	0.045	0.044
6	0.013	0.019	0.030	0.043	0.058	0.067	0.093	0.096
8	0.020	0.036	0.043	0.075	0.102	0.150	0.217	0.219
10	0.026	0.060	0.081	0.135	0.179	0.200	0.325	0.385

$k = 2$

(c) Makespan gap

m \ n	15	20	25	30	35	40	45	50
2	0.14	0	0.08	0.06	0	0.05	0.04	0.04
4	0.12	0.22	0.21	0.06	0.17	0.09	0.04	0
6	4.87	1.11	2.59	1.49	0.53	0.37	0.24	0.30
8	32.00	7.36	0.85	0.95	0.78	0.76	0.42	0.76
10	65.48	7.61	8.24	3.38	2.89	1.53	0.56	0.83

(d) Run time

m \ n	15	20	25	30	35	40	45	50
2	0.02	0.05	0.05	0.07	0.11	0.45	0.72	1.09
4	0.18	0.78	0.64	0.73	2.40	4.68	7.40	5.21
6	0.50	0.87	3.27	2.82	8.53	19.86	11.44	29.11
8	1.02	3.38	8.65	17.52	21.71	42.08	49.94	94.56
10	2.30	5.59	10.94	23.56	59.58	53.83	100.87	195.12

$k = 3$

(e) Makespan gap

m \ n	15	20	25	30	35	40	45	50
2	0.14	0.06	0	0.02	0.07	0.82	0.09	0.04
4	0.20	0.41	0.14	0.12	0.20	0.08	0.08	0.11
6	2.45	1.65	0.56	0.12	0.10	0.16	0.08	0.04
8	12.88	4.13	0.42	0.37	0.96	0.42	0.40	0.41
10	17.59	2.27	3.09	1.10	0.86	0.93	0.40	0.26

(f) Run time

m \ n	15	20	25	30	35	40	45	50
2	0.09	0.50	1.12	1.72	4.00	11.55	17.01	28.55
4	1.21	7.75	14.09	34.79	56.88	121.77	291.35	505.66
6	7.90	25.3	64.16	131.15	244.04	980.48	1262.60	2590.84
8	19.78	80.02	194.86	435.55	987.58	2624.76	3264.98	5114.31
10	54.73	153.17	347.68	604.12	2002.52	4088.55	6251.35	15462.72

TABLE 2: GLS Experiments: $k = 1, 2, 3$, using a random initial solution. The makespan gap is given in % and the run time in seconds.

GLS Experiments

Random initial solution

(a) Makespan gap

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.56	0.06	0.05	0.005	0.02	0.02	0.02	0.01	0.02	0.01
4	6.23	0.54	0.22	0.10	0.10	0.05	0.02	0.04	0.02	0.02
6	17.52	1.05	0.36	0.29	0.15	0.15	0.07	0.09	0.10	0.07
8	66.76	5.64	1.12	0.50	0.27	0.37	0.17	0.12	0.14	0.15
10	86.32	9.86	2.13	0.87	0.64	0.33	0.26	0.32	0.27	0.20

(b) Run time

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.003	0.02	0.11	0.19	0.54	1.78	4.51	5.66	7.93	14.52
4	0.04	0.23	0.84	2.22	4.81	8.02	19.77	62.60	82.57	131.87
6	0.78	0.80	3.09	9.10	25.45	51.87	116.10	206.22	322.23	456.87
8	10.07	5.73	13.29	21.49	61.11	97.83	198.89	427.29	640.75	1117.00
10	32.48	23.80	39.58	98.76	185.58	360.85	659.68	533.94	1154.13	1614.11

GMS initial solution

(c) Makespan gap

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.52	0.06	0.05	0.005	0.02	0.02	0.02	0.01	0.02	0.01
4	6.44	0.35	0.16	0.08	0.07	0.03	0.02	0.04	0.02	0.02
6	17.52	1.54	0.57	0.29	0.17	0.07	0.10	0.10	0.07	0.07
8	66.76	6.27	0.85	0.57	0.36	0.27	0.14	0.12	0.09	0.09
10	86.32	9.75	2.33	1.07	0.45	0.36	0.26	0.34	0.21	0.17

(d) Run time

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.002	0.02	0.11	0.17	1.17	0.92	3.56	6.89	14.87	11.88
4	0.03	0.16	0.56	1.90	3.43	7.78	10.23	58.41	60.20	153.82
6	0.88	1.12	3.21	6.83	20.65	47.00	99.30	166.63	203.52	209.51
8	5.42	8.80	23.13	15.65	38.21	62.50	175.91	530.53	602.13	1157.53
10	23.73	35.67	37.07	84.23	167.81	399.51	345.57	697.98	1457.24	1588.73

TABLE 3: Experiments GLS: k=2, using random and GMS initial solution. The makespan gap is given in % and the run time in seconds.

Full results for VDS experiments

VDS Experiments

Random initial solution

(a) Makespan gap

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.42	0.06	0.05	0.04	0.01	0.01	0.01	0.01	0.01	0.01
4	6.15	0.45	0.20	0.09	0.06	0.05	0.03	0.03	0.04	0.03
6	16.67	2.03	0.76	0.36	0.17	0.16	0.11	0.09	0.09	0.06
8	39.52	4.38	1.66	0.98	0.45	0.27	0.27	0.15	0.17	0.14
10	74.41	10.72	2.95	1.52	0.81	0.71	0.44	0.49	0.33	0.30

(b) Run time

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.01	0.10	0.38	1.04	2.33	4.48	7.89	12.97	19.78	23.71
4	0.03	0.33	0.96	2.8	5.16	10.38	17.77	28.79	45.04	68.83
6	0.06	0.80	3.18	8.09	15.77	27.5	47.15	79.88	113.68	163.41
8	0.11	2.08	6.33	13.37	32.37	59.61	108.79	164.35	249.3	395.79
10	0.29	4.65	22.51	57.23	110.67	184.46	364.6	508.84	416.92	550.51

GMS initial solution

(c) Makespan gap

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.30	0.06	0.05	0.04	0.01	0.01	0.01	0.01	0.01	0.01
4	5.98	0.48	0.17	0.09	0.06	0.05	0.03	0.03	0.04	0.03
6	16.67	1.02	0.48	0.29	0.17	0.10	0.10	0.07	0.06	0.03
8	39.52	4.40	1.31	0.54	0.40	0.27	0.20	0.17	0.08	0.09
10	74.41	10.62	1.82	0.97	0.87	0.57	0.40	0.34	0.22	0.21

(d) Run time

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.005	0.04	0.12	0.36	0.73	1.66	2.73	4.12	6.89	8.02
4	0.02	0.18	0.73	1.91	3.73	7.58	14.25	16.9	30.05	47.58
6	0.03	0.59	2.12	4.92	10.35	19.15	35.08	53.96	83.02	137.75
8	0.05	0.78	4.02	9.29	21.99	36.30	82.26	110.99	168.01	234.97
10	0.08	0.69	7.07	16.36	28.4	60.31	99.14	189.61	229.96	339.72

TABLE 4: Experiments VDS: k=2, using random and GMS initial solution. The makespan gap is given in % and the run time in seconds.

Full results for simulated annealing experiments

Simulated Annealing: Initial Temperature Experiments

m \ n	10	20	30	40	50	60	70	80	90	100
2	137.50	148.50	151.58	155.20	158.34	159.77	160.23	160.42	161.75	162.98
4	125.65	132.39	132.44	135.34	137.76	135.69	136.86	136.74	137.98	137.36
6	112.59	111.42	111.93	113.47	112.62	112.60	109.90	108.87	109.27	107.83
8	99.38	95.47	98.13	94.30	94.06	93.97	91.72	91.25	91.70	90.41
10	90.92	87.78	84.77	82.27	83.51	82.64	81.23	79.21	79.73	77.77

TABLE 5: Simulated Annealing Experiments for determining the initial temperature, using the algorithm by Ben-Ameur [2004]. The temperature values are shown in the table and are the average result of 100 runs of the algorithm per combination of n and m.

Simulated Annealing Experiments with Different Cooling Schedules

Exponential Multiplicative

(a) Makespan gap

m \ n	10	20	30	40	50	60	70	80	90	100
2	1.53	0.44	0.29	0.21	0.15	0.16	0.10	0.12	0.08	0.04
4	15.02	5.22	5.47	3.22	2.03	1.93	1.80	1.46	1.44	2.00
6	27.12	14.37	10.21	10.67	8.08	7.04	5.91	5.27	4.90	4.37
8	49.03	26.15	18.73	16.85	14.32	10.54	10.74	11.70	9.04	11.03
10	74.41	46.23	27.69	23.50	20.27	22.58	14.06	19.23	16.68	18.97

(b) Run time

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.004	0.005	0.006	0.007	0.008	0.009	0.01	0.011	0.012	0.013
4	0.004	0.006	0.007	0.008	0.009	0.009	0.011	0.012	0.013	0.014
6	0.004	0.005	0.007	0.008	0.009	0.010	0.011	0.012	0.013	0.014
8	0.004	0.006	0.007	0.008	0.009	0.010	0.012	0.013	0.015	0.015
10	0.004	0.006	0.007	0.008	0.009	0.010	0.011	0.012	0.013	0.014

Simple Exponential

(c) Makespan gap

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.62	0.16	0.13	0.15	0.1	0.06	0.05	0.07	0.03	0.03
4	13.61	9.11	5.94	3.15	4.29	3.77	3.10	3.01	2.10	2.71
6	35.97	20.15	15.51	10.14	11.13	10.49	9.77	7.75	6.81	7.18
8	48.48	32.54	27.04	21.54	20.39	19.17	16.95	16.88	13.25	14.84
10	76.40	53.07	34.71	35.65	27.68	27.96	25	25.57	19.42	20.73

(d) Run time

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.005	0.007	0.009	0.011	0.012	0.014	0.016	0.017	0.019	0.022
4	0.006	0.008	0.010	0.012	0.013	0.015	0.017	0.019	0.021	0.022
6	0.005	0.008	0.010	0.012	0.014	0.017	0.019	0.020	0.022	0.024
8	0.006	0.008	0.011	0.012	0.015	0.017	0.019	0.021	0.022	0.024
10	0.006	0.008	0.010	0.013	0.015	0.017	0.018	0.020	0.022	0.024

TABLE 6: Simulated Annealing Experiments, using different cooling schedules with a random initial solution. The makespan gap is given in % and the run time in seconds.

Linear Multiplicative

(a) Makespan gap

m \ n	10	20	30	40	50	60	70	80	90	100
2	1.68	0.50	0.23	0.17	0.08	0.13	0.07	0.10	0.06	0.05
4	13.57	5.67	3.60	2.09	2.10	1.84	1.52	1.96	1.80	1.36
6	24.51	14.50	9.96	7.34	6.71	6.97	6.38	6.45	5.41	6.64
8	42.67	29.03	22.98	15.91	13.85	12.43	12.65	12.84	12.94	8.49
10	75.94	43.49	27.51	27.18	22.34	22.40	21.44	17.39	18.47	18.14

(b) Run time

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.004	0.005	0.006	0.007	0.008	0.009	0.010	0.011	0.011	0.013
4	0.003	0.005	0.006	0.007	0.008	0.009	0.010	0.011	0.012	0.013
6	0.004	0.005	0.006	0.007	0.009	0.010	0.011	0.012	0.013	0.014
8	0.004	0.005	0.007	0.008	0.009	0.010	0.011	0.012	0.013	0.015
10	0.004	0.006	0.007	0.008	0.009	0.010	0.011	0.012	0.013	0.014

Quadratic Multiplicative

(c) Makespan gap

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.33	0.06	0.05	0.04	0.01	0.01	0.01	0.01	0.01	0.01
4	6.06	0.37	0.28	0.13	0.11	0.08	0.08	0.05	0.05	0.03
6	16.67	1.66	0.97	0.85	0.45	0.29	0.27	0.33	0.18	0.19
8	39.52	4.79	2.59	1.65	1.16	1.03	0.71	0.72	0.64	0.40
10	74.41	11.17	4.76	3.23	2.22	1.93	1.23	1.08	0.89	0.81

(d) Run time

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.43	0.62	0.78	0.93	1.04	1.18	1.33	1.49	1.57	1.74
4	0.46	0.64	0.78	0.93	1.11	1.24	1.40	1.56	1.72	1.87
6	0.49	0.65	0.85	1.02	1.14	1.31	1.44	1.58	1.74	1.85
8	0.51	0.68	0.85	1.00	1.16	1.32	1.46	1.64	1.75	1.89
10	0.52	0.71	0.87	1.02	1.18	1.35	1.50	1.65	1.80	1.95

TABLE 7: Simulated Annealing Experiments, using the quadratic multiplicative cooling schedule with a random initial solution. The makespan gap is given in % and the run time in seconds.

Simulated Annealing Experiments

Random initial solution

(a) Makespan gap

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.33	0.06	0.05	0.04	0.01	0.01	0.01	0.01	0.01	0.01
4	6.06	0.37	0.28	0.13	0.11	0.08	0.08	0.05	0.05	0.03
6	16.67	1.66	0.97	0.85	0.45	0.29	0.27	0.33	0.18	0.19
8	39.52	4.79	2.59	1.65	1.16	1.03	0.71	0.72	0.64	0.40
10	74.41	11.17	4.76	3.23	2.22	1.93	1.23	1.08	0.89	0.81

(b) Run time

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.43	0.62	0.78	0.93	1.04	1.18	1.33	1.49	1.57	1.74
4	0.46	0.64	0.78	0.93	1.11	1.24	1.40	1.56	1.72	1.87
6	0.49	0.65	0.85	1.02	1.14	1.31	1.44	1.58	1.74	1.85
8	0.51	0.68	0.85	1.00	1.16	1.32	1.46	1.64	1.75	1.89
10	0.52	0.71	0.87	1.02	1.18	1.35	1.50	1.65	1.80	1.95

GMS initial solution

(c) Makespan gap

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.42	0.06	0.05	0.04	0.01	0.01	0.01	0.01	0.01	0.01
4	6.06	0.30	0.24	0.16	0.11	0.08	0.07	0.03	0.05	0.03
6	16.67	2.05	1.19	0.49	0.34	0.26	0.24	0.22	0.14	0.16
8	39.52	4.78	2.54	1.08	0.75	0.72	0.51	0.37	0.37	0.23
10	74.41	10.62	4.37	2.50	1.32	1.35	0.75	0.7	0.57	0.40

(d) Run time

m \ n	10	20	30	40	50	60	70	80	90	100
2	0.43	0.62	0.81	0.89	1.04	1.19	1.31	1.52	1.56	1.77
4	0.46	0.68	0.84	1.02	1.19	1.35	1.40	1.50	1.67	1.75
6	0.50	0.68	0.84	1.04	1.16	1.32	1.46	1.54	1.66	1.87
8	0.55	0.73	0.87	1.06	1.20	1.34	1.51	1.70	1.83	1.98
10	0.55	0.74	0.90	1.04	1.22	1.39	1.58	1.76	1.91	2.04

TABLE 8: Experiments Simulated Annealing: k=2, using random and GMS initial solution. The makespan gap is given in % and the run time in seconds.

Results of Figure 9

Poor Distribution

(a) Makespan gap

m \ n	10^2	5×10^2	10^3	5×10^3	10^4	5×10^4
2	0.127066	0.009453	0.002206	0.000137	0.000042	0.000004

(b) Run time

m \ n	10^2	5×10^2	10^3	5×10^3	10^4	5×10^4
2	0.59	5.65	15.59	184.36	509.67	5185.65

Uniform Distribution

(c) Makespan gap

m \ n	10^2	5×10^2	10^3	5×10^3	10^4	5×10^4
2	0.103995	0.00808	0.001029	0.00006	0.000023	0.000003

(d) Run time

m \ n	10^2	5×10^2	10^3	5×10^3	10^4	5×10^4
2	0.42	4.52	12.98	155.38	482.84	4521.27

TABLE 9: Testing the limits of the simulated annealing algorithm, using the GMS initial solution