Empirical analysis of procedures that schedule unit length jobs subject to precedence constraints forming in- and out-stars

Samuel Tigistu Feder*

Abstract

This paper addresses the problem of scheduling unit length jobs subject to precedence constraints forming in- and out-stars. There are several different procedures that can be used to solve this problem and our aim in this paper is to compare these procedures empirically. We found that one algorithm that was thought to solve the problem optimally didn't do so for the general set of instances and that all procedures that we tested did not perform well if the number of stars in the instance was close but below the number of machines in the instance.

1 Introduction to the problem

The problem that underlies this paper is a variant of the job shop scheduling problem. To better understand the idea behind the job shop scheduling problem imagine a situation in which a company with a certain number of identical machines m wants to schedule different jobs $j \in J$ on these machines in such a way that the total makespan is minimized. Put differently, the last job should be done as early as possible. For visualization imagine an engineer that has to watch the machines and can only go home once all jobs are finished. In the job shop scenario running several machines at the same time does not incur any extra cost. However every extra unit of time that the engineer has to stay is costly and should be minimized. There exists a large amount of literature about different variants of this problem. In the group of variants

* Samuel Tigistu Feder received a bachelor degree in Econometrics and Operations at Maastricht University in 2014. At the moment he takes a Master in Econometrics and Operations with a specialisation in Operations research at Maastricht University
Contact: s.feder@student.maastrichtuniversity.nl

that we are dealing with all jobs take exactly 1 unit of time to process on any of the machines. Furthermore the jobs have to be scheduled subject to precedence constraints. Thus some jobs cannot be started before certain other jobs are finished. More formally, any instance in this group of variants of the problem can be described completely by the number of machines m and an acyclic directed graph $D = (J, A)$. This graph is called the precedence graph and the nodes in the graph correspond to the different jobs while any directed arc $(i, j) \in A$ in the graph corresponds to a precedence constraint and implies that job i has to be finished before job j can be started. Of course $D = (J, A)$ cannot have a directed cycle because if the precedence constraint where forming a cycle the problem would be infeasible. As an example, consider a situation where job 1 cannot be started before job 2 is finished, and job 2 cannot be started before job 1 is finished. In this case, none of the jobs can ever be started. The general definition of an acyclic directed graph does however allow for non-directed cycles. Thus for example an instance in which job 1 has to be done before job 2 and job 3 can be started and job 2 has to be done before job 3 can be started is allowed. An important feature of a precedence graph is its height h, which is defined as the the number of arcs in the longest directed path in the graph. Please note that if there were not any precedence constraints, thus if $A = \emptyset$, the problem becomes trivial since simply scheduling all jobs as evenly as possible would be feasible and optimal, resulting in an optimal makespan of $\lceil |J| / m \rceil$. Therefore $\lceil |J| / m \rceil$ also is a lower bound for the optimal makespan whenever $A \neq \emptyset$. Since the general problem of scheduling unit length jobs subject to precedence constraints is NP-hard (Ullman, 1975) most academic literature focuses on special variants of this problem by imposing certain restrictions on the shape of the precedence graph. In the particular variant of the problem that we will be dealing with in this paper we will be imposing the restriction that the precedence constraints in $D = (J, A)$ have a form which we will call in-stars an out-stars. Formally, an in-star is a star for which the out-degree of all nodes in the star is ≤ 1 while an out-star is a star for which the in-degree of all nodes that form the star is ≤ 1. We will be denoting the sum of the number of in- and out-stars by k. Note that this implies that all graphs that we consider have a maximal height of 2. For a less formal but more intuitive description of our problem start by imagining k different sets of jobs. Each of these sets of jobs has exactly one center-job. A set of jobs forms an in-star if there are either one or zero jobs that have to done before the center-job can be started and if there is an arbitrary number of jobs that can be done in any order after the center-job is finished. Similarly, a set of jobs forms an out-star if there is an arbitrary number of jobs that has to be done in any order before the center-job can be started and if there are either one or zero jobs that can only be started once the center-job is finished. Please also consider the examples in Figure 1. In this figure each connected set of circles, crosses and discs represents a star, each circle represent an in-job, each cross represents a center-job and each disc represents an out-job. Here the

precedence arcs are represented by lines instead of arrows because all arrows would have only pointed upwards. Note that a star that has one or zero in-jobs as well as one or zero out jobs is both an in- and out-star at the same time.

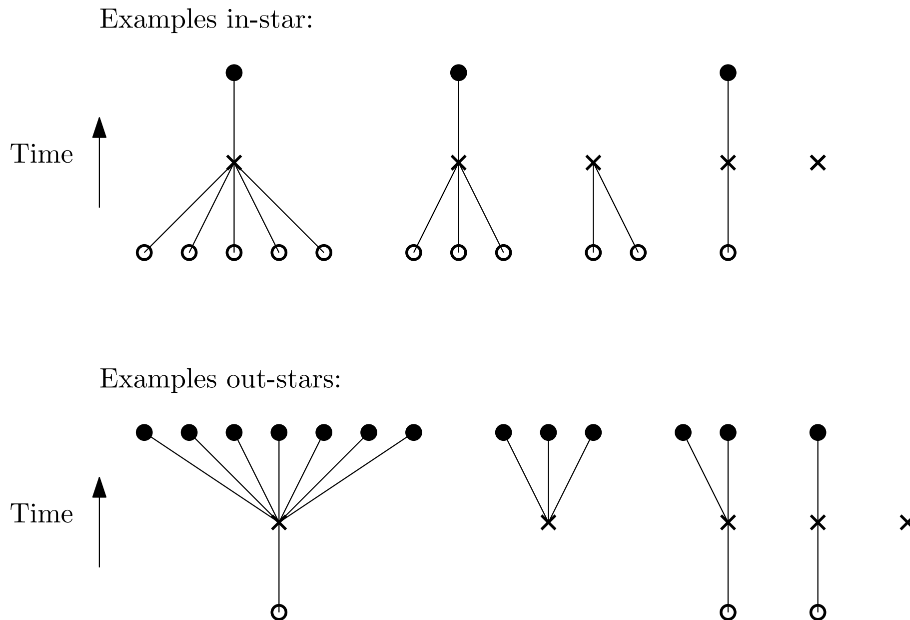


Figure 1: An example illustrating the concept of in- and out-stars

There exist an algorithm that is polynomial in $|J|$ that can schedule any variant of the problem in which the precedence constraints form forests (Hu, 1961). Thus since stars are trees this algorithm also works for our variant of the problem, if the the total number of jobs $|J|$ is taken as a measure of the input size. However, since in our variant of the problem we already know the form of the precedence constraints it is possible for us to write the input more compactly by simply writing down the number of in- and out jobs for each star. This reduces the input size of the problem to the number of stars k. In (Berger et al., 2014) it is shown that if we take the input size to be k the the problem is NP-complete if not only in- and out-stars but any kind of star is allowed. If the restriction is to only use in- and out- stars (Berger et al., 2014) proposes six different polynomial time procedures to find a solution to the problem. The first five procedures are heuristics that have a worst case complexity of $O(k * log(k))$ and the sixth procedure is an algorithm that should find an optimal solution and has a worst case complexity of $O(k^2)$. Our aim in this paper is to empirically compare these six different procedures with respect to their performance and their running time. One of the main findings in this paper is that the sixth procedure does not always find an optimal solution for instances with general in- and out-stars. The data we collected suggests that the sixth procedure is only guaranteed to find an

optimal solution if all in- and out-stars of the instance have at least one in-job and at least one out-job. In terms of our earlier description this means that the sixth procedure is only guaranteed to find an optimal solution if for each center-job there is at least one job that has to be done before the center-job can be started and there is at least one job that can only be started if the center-job is already finished. In order to conduct a meaningful comparison we think that it is necessary to know the actual best possible solution for all the instances on which we ran the different procedures. Thus we designed a linear program using cplex that is guaranteed to always find the best possible solution. It is important to note that the performance of the other algorithms still remains very relevant since even for small instances the linear program is by several orders of magnitude slower than the other procedures. From now on we will refer to the first five procedures as Heuristic 1-5, we will refer to the sixth procedure as Algorithm 1 and we will refer to the linear programming procedure as Linear program.

2 Literature review

We found several texts dealing with the scheduling of unit length jobs subject to precedence constraints. However most of them measured input size in terms of the total number of jobs in the instance $|J|$. Therefore it is a somewhat difficult to compare them to the problem that we are trying to solve. As already mentioned the general problem is NP-complete (Ullman, 1975) and if the precedence constraints form a forest there exist an efficient algorithm (Hu, 1961). But even if the precedence constraints do not form a forest an efficient algorithm exists as long as some other criteria are met. A lot of the NP-completeness proves for the general problem and some of its variants need the number of machines m to grow as $|J|$ grows and Dolev and Warmuth (1984) shows that if both m and the height of the precedence graph h are bounded by m^* and h^* then there exist an $O(|J|^{h^*(m^*-1)+1})$ algorithm that solves the problem. We found this particularly interesting because it implies that there exists a strong linkage between the number of machines m and the amount of work required to find an optimal solution. Another example of a variant of the general problem for which efficient algorithms are known is the situation in which $D=(J,A)$ is a graph whose incomparability graph is chordal (Papadimitriou and Yannakakis, 1979). The incomparability graph $G=(J,E)$ of $D=(J,A)$ is defined as an undirected graph with $[i,j]\in E$ if and only if $(i,j),(j,i)\notin A$. This means that all jobs in graph D that have no direct precedence relation are connected with an undirected edge in graph G, and all the jobs that have a direct precedence relation in D are unconnected in G. The graph $G=(J,E)$ is chordal if and only if for every cycle in G that contains at least 4 vertices there exists an edge between two of the vertices in the cycle, called a chord, that is not part of the cycle. Also consider the examples in Figure 2.

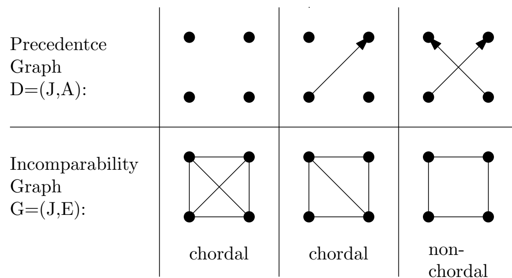


Figure 2: Ilustrating chordal incomparability graphs

For the algorithm in (Papadimitriou and Yannakakis, 1979) to work efficiently it is however implicitly assumed that the graph $D = (J, A)$ is already given in its transitively closed form, Gabow (1981) showed that this assumption is not necessary in order to find a solution efficiently.

3 Introduction to the procedures

We will use the same basic representation of an instance as in (Berger et al., 2014) in order to describe how the algorithms work. Let $D = S_1, ..., S_k$ be the set of all stars and for $1 \leq i \leq k$ let A_i be the set of in-jobs of star i, thus the set of jobs that have to be done before the center-job can be started. Similarly for $1 \leq i \leq k$ let B_i be the set of out-jobs of star i, thus the set of jobs that can only be started once the center-job is finished. Let A and B correspond to the union over all i of A_i and B_i respectively. Also please note that one can trivially construct a feasible solution by first scheduling all in-jobs in the first $\lceil |A| / m \rceil$ time slots, then scheduling all centers in the next $\lceil k/m \rceil$ time slots and finally schedule all out-jobs in the last $\lceil |B| / m \rceil$. Thus it is possible to schedule all jobs in $\lceil |A| / m \rceil + \lceil |k| / m \rceil + \lceil |B| / m \rceil$ time slots. Since

$$\begin{aligned} &\lceil |A| / m \rceil + \lceil k/m \rceil + \lceil |B| / m \rceil \\ &\leq \lceil (|A| + k)/m \rceil + 1 + \lceil |B| / m \rceil \\ &\leq \lceil (|A| + k + |B|)/m \rceil + 2 \\ &= \lceil |J| / m \rceil + 2 \end{aligned}$$

we should expect all reasonable procedures that attempt to solve the problem to always find a solution whose makespan lies within $\lceil |J| / m \rceil$ and $\lceil |J| / m \rceil + 2$. It is instructive to be aware of the fact that the data that was available to each procedure about a particular instance of the problem is simply k, m and $|A_i|$ and $|B_i|$ for all $1 \leq i \leq k$. Furthermore note that if outputting a schedule requires printing for every single job the time slot it occupies in the schedule, it would not be possible to have a procedure whose running time is polynomial in k since the output alone would take $O(|J|)$ time. This is the reason why the algorithms that we are discussing simply report the position of the centers and the total length. Finding a feasible schedule for the in- and out-start scheduling problem after the position of the centers are known is trivial.

3.1 Heuristics 1-5

Heuristics 1-5 are as already mentioned taken from (Berger et al., 2014). Each of the five heuristics is characterized by a certain priority rule with respect to the center-job of the stars:

1. Largest ratio of $\frac{out-degree}{in-degree} = \frac{|A_i|}{|B_i|}$ first
2. Largest $out - degree = |B_i|$ first
3. Largest $in - degree = |A_i|$ first
4. Largest ratio of $\frac{in-degree}{out-degree} = \frac{|B_i|}{|A_i|}$ first
5. Smallest $in - degree = |A_i|$ first, ties broken by largest $out - degree = |B_i|$ first

$In - degree$ and $out - degree$ are properties of the center-job, while $|A_i|$ and $|B_i|$ are properties of the corresponding star, but since each star has exactly one center we can use them interchangeably. Since stars with zero in-degree and/or zero out-degree exist, the priority rules 1 and 4 are not always defined. Thus we decided to treat the ratio of a positive number and zero as positive infinite and scheduled all pure centers, those with $|A_i| = |B_i| = 0$, at the beginning for Heuristic 1 and at the end for Heuristic 4. Once the stars are ordered according to the priority rule, the heuristics greedily schedules the centers as early as possible in the given order. Finally, once all in-jobs and center-jobs are scheduled the heuristics greedily schedule the out-jobs. Since the greedy scheduling of each star can be done in constant time, the scheduling part of the heuristics has a computational complexity of $O(k)$. However, because the sorting part has a computational complexity of $O(k * log(k))$, the computational complexity of the whole procedure is $O(k * log(k))$.

3.2 Algorithm 1

Algorithm 1 is the main algorithm in (Berger et al., 2014) that deals with the problem at hand. It also starts by ordering the stars, using the same condition that Heuristic 5 uses: Smallest $in - degree = |A_i|$ first, ties broken by largest $out - degree = |B_i|$ first. It is based on Lemma 7 in (Berger et al., 2014) which we will repeat here as Lemma 1 without proof. Let $s[i]$ be the minimum makespan needed to schedule all jobs in A_i and their corresponding centers c_i for $1 \leq i \leq k$ in the just described order. Let $f[i]$ be the amount of idle time in such a schedule. Observe that $f[i] = m * s[i] - (i + \sum_{j=1}^{i} A_j)$ where in the brackets i represents the centers scheduled and the sum of A_j represents the in-jobs scheduled. Define $s[0] = f[0] = 0$. Then:

Lemma 1 *All values $s[1],...,s[k]$ and the center placements in the corresponding schedules can be computed altogether in total time $O(k^2)$ using the following formula:*

For $1 \leq i \leq k$, $s[i] = min_{0<l<i}\left\{s[i-l] + max\left\{0, \left\lceil\frac{\sum_{j=i-l+1}^{i}|A_j| - f[i-l]}{m}\right\rceil\right\} + \left\lceil\frac{l}{m}\right\rceil\right\}$

Algorithm 1 starts by calculating $s[i]$ and $f[i]$ for $1 \leq i \leq k$. Creating preliminary schedules as follows: Consider schedule i , and let l^* be the value of l for which the term in Lemma 1. is minimal

$l^* = argmin_{0<l<i}\left\{s[i-l] + max\left\{0, \left\lceil\frac{\sum_{j=i-l+1}^{i}|A_j| - f[i-l]}{m}\right\rceil\right\} + \left\lceil\frac{l}{m}\right\rceil\right\}$.

Then centers c_1,...,c_{i-l^*} are scheduled in the same way as they are scheduled in the already computed schedule $i - l^*$ while centers c_{i-l^*+1},...,c_i are scheduled in the last $\lceil\frac{l}{m}\rceil$time slots of schedule i.

Then, for each preliminary schedule i Algorithm 1:

1. Greedily inserts jobs A_{i+1}...A_k into the schedule by first using up idle time $f[i]$.
2. Defines t to be the first time slot strictly after $s[i]$ that has idle time and defines q to be the amount of idle time in time slot t.
3. Schedules as many centers $c_{i+1},...,c_{i+q-1}$as possible in time slot t, and the remaining centers greedily in time slot $t+1$ and further.
4. Greedily schedules jobs from B_1,...,B_k starting at time slot t.

This gives us k different schedules. Algorithm 1 now simply chooses the schedule with the shortest makespan out of those k and presents this schedule as its solution. According to Berger et al. (2014) the whole algorithms has a computational complexity of $O(k^2)$. Take into account that the centers in the schedules that Algorithm 1 produces will always be scheduled in the same order as the centers in the schedules that Heuristic 5 produces. However the centers might and often will be scheduled in different time slots because it is often beneficial to delay the processing of some of the centers that Heuristic 5 schedules early in order to process in-jobs of other stars first.

3.3 Linear program

The linear program that we set up in order to solve each instance optimally is the following:

$\min max$
subject to
$x^t * t \leq max$ for $1 \leq t \leq T$

$\sum_{i=1}^{k}(a_i^0 + c_i^0 + b_i^0) = 0$
$\sum_{i=1}^{k}(a_i^t + c_i^t + b_i^t) \leq m * x^t$ for $1 \leq t \leq T$
$\sum_{t=1}^{T} a_i^t = |A_i|$ for $1 \leq i \leq k$
$\sum_{t=1}^{T} c_i^t = 1$ for $1 \leq i \leq k$
$\sum_{t=1}^{T} b_i^t = |B_i|$ for $1 \leq i \leq k$
$\sum_{j=0}^{t} a_i^t \geq |A_i| * AllAScheduled_i^t$ for $0 \leq t \leq T$ and $1 \leq i \leq k$
$\sum_{j=0}^{t} c_i^t \geq AllCScheduled_i^t$ for $0 \leq t \leq T$ and $1 \leq i \leq k$
$c_i^t \leq AllAScheduled_i^{t-1}$ for $0 \leq t \leq T$ and $1 \leq i \leq k$
$b_i^t \geq |B_i| * AllCScheduled_i^{t-1}$ for $0 \leq t \leq T$ and $1 \leq i \leq k$

Where we have the following data:

- $|A_i|$ is the number of in-jobs belonging to center i
- $|B_i|$ is the number of out-jobs belonging to center i
- m is the number of machines in the instance
- k is the number of stars in the instance
- T is a number of time slots that is large enough to ensure that there exist an optimal solution that uses less time slots than T, we use $T = \lceil |J| / m \rceil + 3$

And where we have the following variables:

- a_i^t is a non-negative integer corresponding to the number of in-jobs belonging to star i that are scheduled in time slot t
- c_i^t is a non-negative integer corresponding to the number of centers belonging to star i that are scheduled in time slot t
- b_i^t is a non-negative integer corresponding to the number of out-jobs belonging to star i that are scheduled in time slot t
- $AllAScheduled_i^t$ is a boolean variable that is equal to one if all in-jobs belonging to star i are scheduled before or in time slot t
- $AllCScheduled_i^t$ is a boolean variable that is equal to one if the center belonging to star i are scheduled before or in time slot t
- x^t is a boolean variable that is equal to one if any job no matter what kind is scheduled in time slot t

- max is a non-negative integer corresponding to the highest time slot in which any jobs are scheduled

Note that this is an integer linear problem and that both the number of variables as well as the number of constraints will grow proportionally to $T * k$, which will make it nearly impossible to solve it for very large instances. Because this procedure was by far the slowest procedure, we chose to avoid using it when possible in order to decrease the computation time of the statistics that we present in section 4.2. Thus, we first run Algorithm 1 and Heuristic 1-5 and if any of these procedures found a schedule whose makespan reached the theoretical lower bound of $\lceil |J| /m \rceil$ we did not run the linear program because we already knew the optimal solution. This helped us to significantly increase the speed of the computations.

4 Results

4.1 Particularly interesting instances

Take into account that all instances that we will be discussing in section 4.1 have a number of machines $m = 3$. Thus for each of these instances we can schedule at most three jobs in each time slot. In some but not all cases we where able to find similar instances for different values of m. However most of those instances where more complicated without adding additional value to the discussion, which is the reason why we decided to present only the instances for which $m = 3$.

4.1.1 Algorithm 1 is not always optimal

The first interesting instance that we want to show is a counter example to the claim in (Berger et al., 2014) that Algorithm 1 will always find an optimal solution. As already pointed out in the introduction, our empirical evidence suggests that Algorithm 1 is only guaranteed to produce an optimal schedule if each star has at least one in-job and at least one out-job. If instances with in- and out-stars that have zero in-jobs and/or zero-out jobs are included Algorithm 1 still performs very well compared to the other heuristics, however there are instances where it does not produce an optimal solution and the following instance is one of those. As we already mentioned the schedule output of Algorithm 1 consists for each star only of the time slot its center is in, as well as the total length of the schedule. We reconstructed the instance and both the preliminary and the complete schedules completely from the output of the algorithm in order to make it easier for the reader to follow the steps of the algorithm shown in Figure 3.

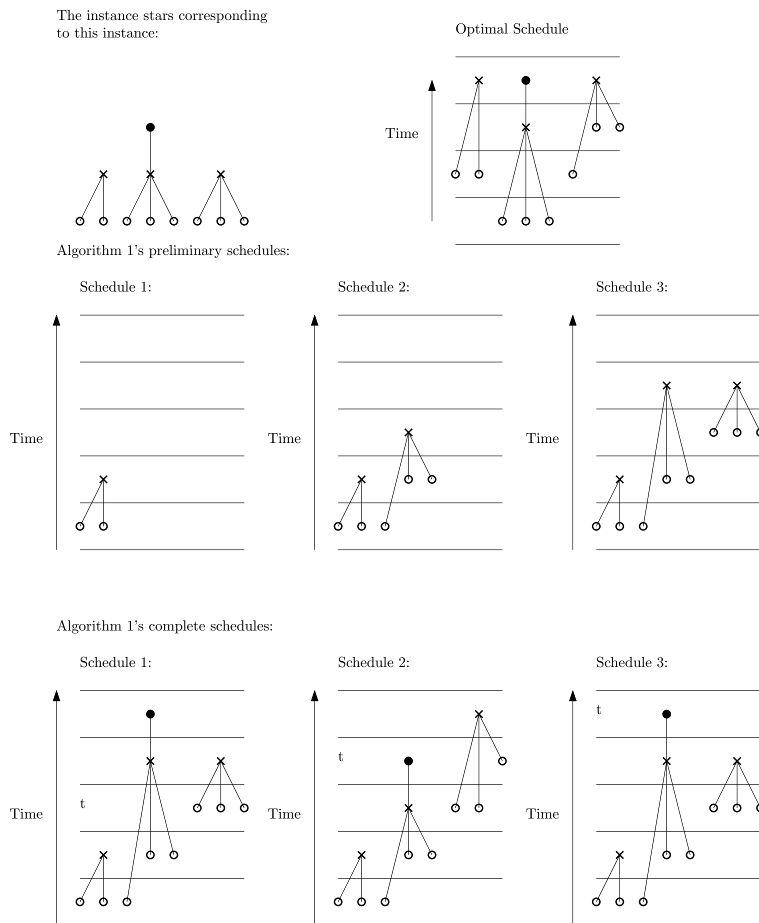


Figure 3: An Example in which Algorithm 1 is not optimal

We also put a t in the time slot for each final schedule that corresponds to the time slot t that is mentioned in the description of Algorithm 1. The following values of $s[i]$ and $f[i]$ correspond to the different preliminary schedules:

i	$s[i]$	$f[i]$
1	2	3
2	3	2
3	4	1

Thus as we can see in Figure 3 Algorithm 1 only finds solutions that have a makespan of five while there exists a feasible solution which has a makespan of four. Therefore Algorithm 1 is not optimal for some instances. We did not find any instance in which the Algorithm 1 would

find a solution whose makespan was by two worse than the optimal makespan. We modified the initial sorting condition that Algorithm 1 uses, in an attempt to make it perform optimally, and although we did not find a sorting condition that was able to achieve this, we did find the particularly interesting instance presented in Figure 4.

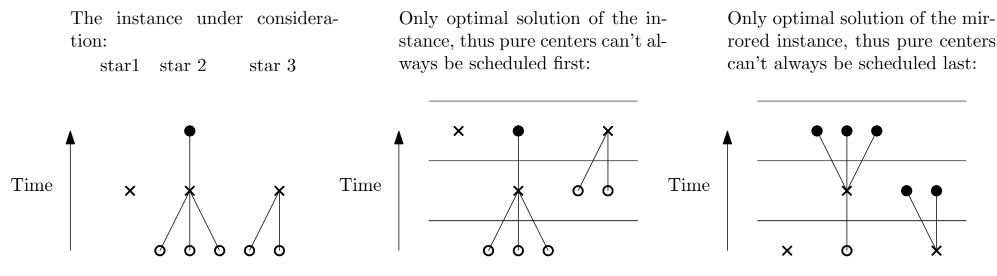


Figure 4: Exaple showing that a change in the sorting rule can't fix Algorithm 1

This instance and its mirror instance both only have a single optimal schedule and together they show that any procedure that either always schedules pure centers at the beginning or always schedules pure centers at the end will never be able to find an optimal solution for all possible instances of our problem, since it will never find the optimal solution of at least one of the two instances that we just presented. Let us explain why for this particular instance there exists only this one particular optimal solution schedule. First of all note that the presented schedule is definitely optimal since it is feasible, the lower bound for the makespan is $\lceil |J| /m \rceil = \lceil 9/3 \rceil = 3$ and the presented schedule has a makespan of three thus reaching the lower bound. Now consider star 2, the only way in which star 2 can fit in a schedule that has a makespan of three is clearly if all its in-jobs are scheduled in the first time slot, its center-job is scheduled in the second time slot and its out-job is scheduled in the third time slot. This leaves two machines unused in both the second and the last time slot. The only way in which star 3 can be scheduled on these still unused machines is by scheduling its two in-jobs in time slot 2 and its center in time slot 3. This leaves only one unused machine in time slot 3 for the pure center that is star 1. Thus as we claimed, since all jobs have to be scheduled exactly in the way our solution describes, the solution is unique. The argument for uniqueness of the optimal solution for the mirrored instance is extremely similar. Thus the solutions are unique and the pure center is not scheduled first in the original instance and not scheduled last in the mirrored instance. Therefore any algorithm that either always schedules pure centers first or always schedules pure centers last will not be able to find an optimal solution for all possible instances of the in- and out-star scheduling problem.

4.1.2 Examples of bad performing heuristics

For Heuristic 1,3,4 and 5 we found instances for which the heuristic would return a solution whose makespan was by 2 worse than the optimal makespan. We will now present the less complex instances for which this was the case. For Heuristic 1 and Heuristic 5 (1. largest ratio of $\frac{out-degree}{in-degree} = \frac{|A_i|}{|B_i|}$ first and 5. smallest $in - degree = |A_i|$ first, ties broken by largest $out - degree = |B_i|$ first) we found the instance in Figure 5 where they both perform bad:

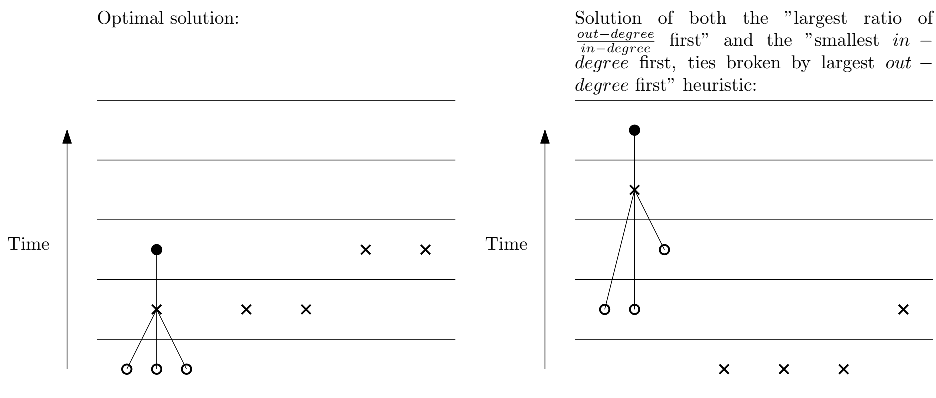


Figure 5: Example in which Heuristic 1 and 5 perform very badly

However for this instance Heuristic 1 only performs bad because of the additional assumption which we made at the beginning that we will schedule pure centers at the beginning for Heuristic 1. For Heuristic 3 and Heuristic 4 (3. largest $in - degree = |A_i|$first and 4. largest ratio of $\frac{in-degree}{out-degree} = \frac{|B_i|}{|A_i|}$first) we found the instance in Figure 6 where they both perform bad:

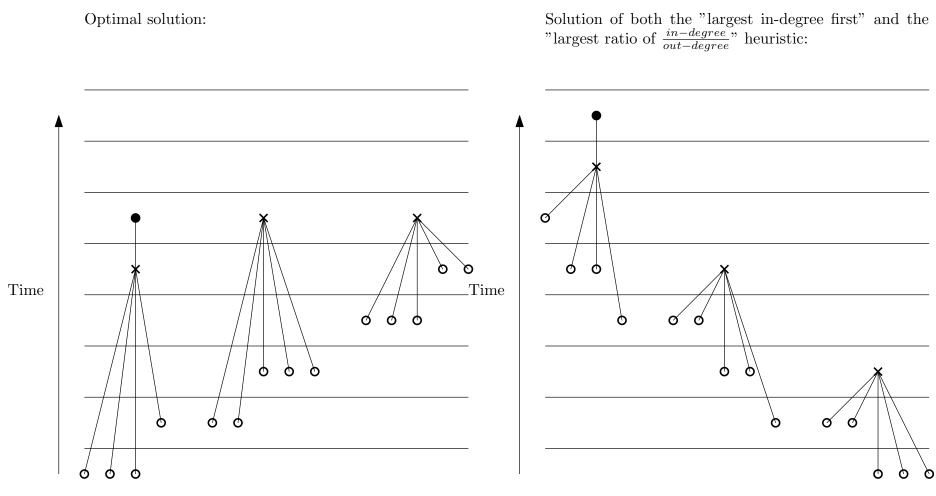


Figure 6: Example in which Heuristic 3 and 4 perform badly

The heuristics were not performing this badly for any instance where each star had at least one in-job and at least one out-job that we tested. For those kind of instances the maximal difference in makespan between the heuristics and the optimal solution was 1, although we do admit that our search for these pathological instances was somewhat limited compared to our search for those pathological instances using general in- and out stars.

4.1.3 No greedy scheduling procedure can be always optimal

For the instance presented in Figure 7 all five heuristics fail to find the optimal solution and since the 3 stars in the instance are identical we know that this failure cannot be attributed to a particular scheduling condition.

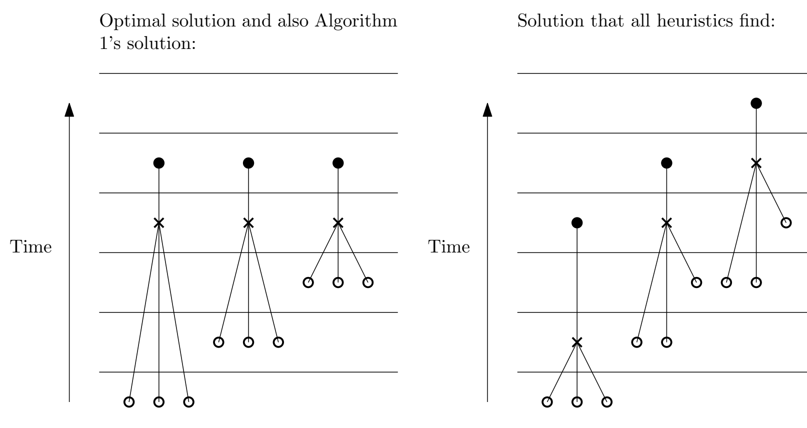


Figure 7: Example in which Algorithm 1 performs better than all Heuristics

The reason why all heuristics fail to find the optimal solution in this case is simply because they all schedule the centers greedily as early as possible and for the instances in Figure 7 no procedure that does this can find the optimal solution. Thus since this instance exist we can conclude that no procedure that greedily schedules centers as early as possible and no combination of results of such procedures can always find the optimal solution for an instance of the in-and out-star scheduling problem. Please note however, that for this instance Algorithm 1 does find the optimal solution. This example instance remains relevant even if we are restricting ourselves to instances where both the in- and out-degree of every in- and out-star is at least one.

4.2 Computation results summary statistics

There are several parameters that could potentially influence the performance of the different procedures, like:

- the maximal number of in- and out-jobs, $max_i |A_i| \leq y$ and $max_i |B_i| \leq y$ for any star in an instance
- the number of stars k in the instance
- the number of machines m in the instance
- how many percentage points of the stars in the instance are in-stars p

There might even be more parameters that could influence the procedures' performance but we decided to focus on the parameters that we just mentioned. In order to be able to make a meaningful assessment of the procedures' performance, given a particular set of values for y,k,m and p, we generated 5000 different instances in the following way:

1. Set the number of machines in the instance equal to m and the number of stars equal to k
2. For each of the k stars randomly decide if it will be an in-star or an out-star, where the probability of an in-star is $p/100$
3. For each in-star the number of in-jobs is a random integer between or equal to zero and y, where each integer has the same probability ($DiscreteUniform[0, y]$) and the number of out-jobs is either zero or one with equal probability ($DiscreteUniform[0, 1]$)
4. Similarly, for each out-star the number of out-jobs is a random integer between or equal to zero and y, where each integer has the same probability ($DiscreteUniform[0, y]$) and the number of in-jobs is either zero or one with equal probability ($DiscreteUniform[0, 1]$)

We want to present how the independent variable performance changes due to changes in the four dependent variables y,k,m and p, thus one simple graph is of course not enough since it only has one axis for a dependent variable and a three dimensional graph would still not be enough and is often hard to read on two dimensional paper. Thus we decided to define two different basic configurations:

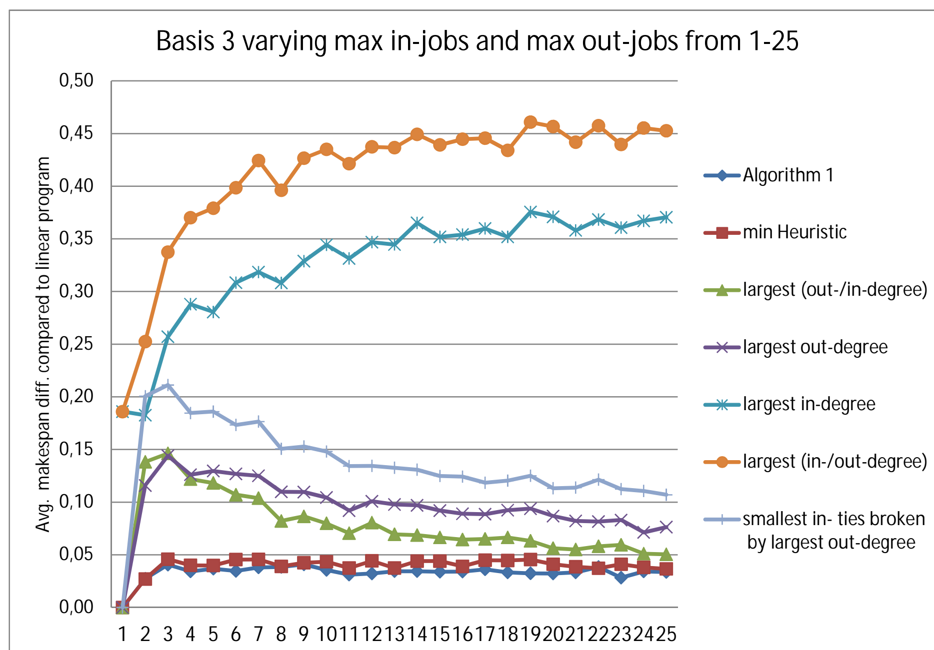
1. Basis 3: $y = k = m = 3$ and $p = 50$
2. Basis 7: $y = k = m = 7$ and $p = 50$

and to make four different graphs of each basis, always varying a different parameter. Finally, we still had to define our measure of performance. We decided to use the average difference between the makespan of the solution for the different procedures and the makespan of the optimal solution for the different instances, thus the average surplus in makespan. We decided to choose the absolute difference instead of taking some percentage difference because as

mentioned in the introduction we should expect all reasonable procedures that attempt to solve the problem to always find a solution whose makespan lies within $\lceil |J| / m \rceil$ and $\lceil |J| / m \rceil + 2$. Since the optimal solution will have a makespan that is larger than or equal to $\lceil |J| / m \rceil$ this implies that we cannot expect the difference between the solution of the different procedures and the optimal solution to grow continuously if we change any of the parameters. Furthermore it did not make sense to us to scale something down by some factor, if we do not expect it to grow by that factor. In our measurement of performance we decided to also have a look at what happens if we combine the knowledge that each of these five heuristics gives us and create a new procedure that simply always reports the result of the best performing heuristic. We called this procedure min Heuristic.

4.2.1 Varying the maximal number of in- and out-jobs

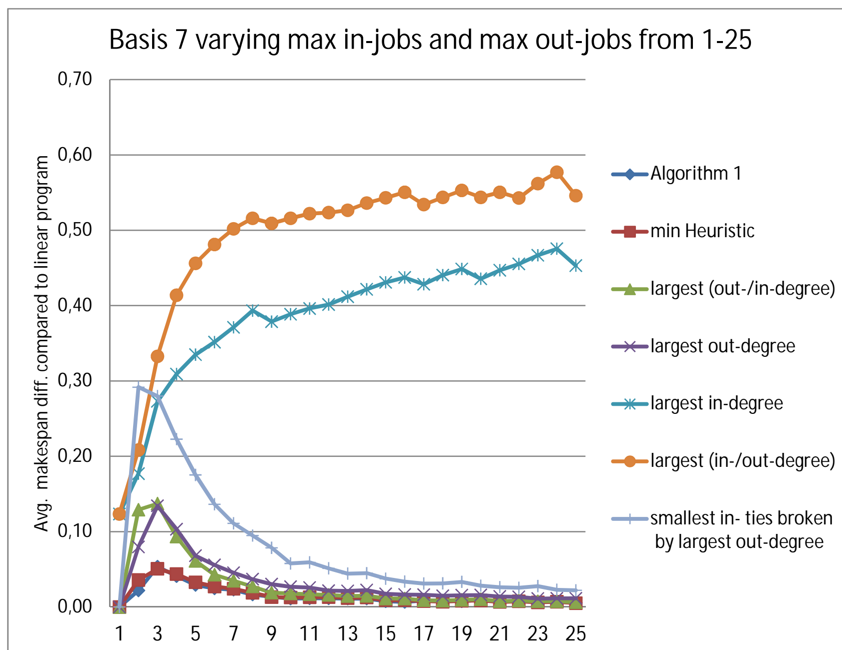
Let us start by having a look at what happens when we vary y the maximal number of in- and out-jobs in a particular instance while keeping the other parameters constant.



Graph 1: Results for Basis 3 when varying max in-jobs and max out-jobs

In Graph 1 we see that both the largest (in/out-degree) first heuristic and the largest in-

degree first heuristic perform quite poorly for most of these instances while the other heuristics perform relatively well. It is also worth noting that all non-linear-programming procedures have a significant decrease in their performance (increase in average surplus makespan) as y changes from one over two to three. Interestingly, Algorithm 1's performance and the performance of the minimum over all five heuristics is very similar. It is also be interesting that the highest sample standard deviation of any surplus makespan was found for the largest (in-/out-degree) heuristics for y larger 16. For those values of y the sample standard deviation of the 5000 differences between the optimal solution makespan and the makespan of the solution of the largest (in-/out-degree) heuristic was around 0.53. In general, the values of the sample standard deviation were relatively closely linked to the reported sample means which is why we chose not to report them separately. Most measured sample standard deviations range from 0.2-0.4 which (given that we have 5000 different observations) makes us quite confident that the reported sample means of the performance in the graph will be very close to their population means. Another interesting outcome that we want to report is the fact that the computation of the linear programming solution dropped significantly as y increases (from 1156/5000 for $y = 1$ to 71/5000 for $y = 25$) where the largest percentage drops occurred for the smaller values of y. This suggests that for large values of the max-in and max out-jobs the probability than one of non-linear-program the procedures found a solution with a makespan that was equal to the theoretical lower bound of $\lceil |J| / m \rceil$ increased significantly.

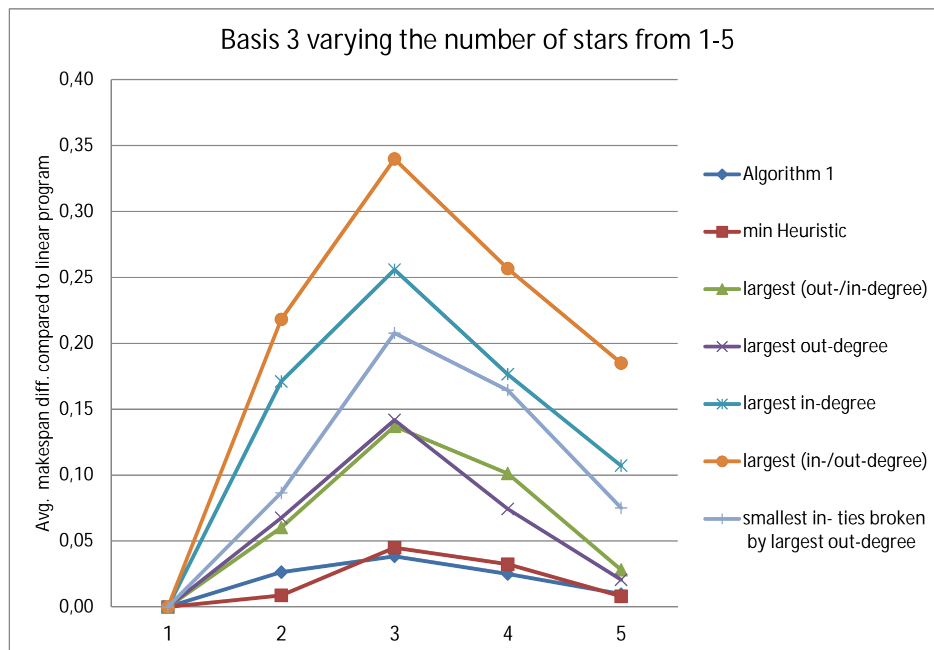


Graph 2: Results for Basis 7 when varying max in-jobs and max out-jobs

The results in Graph 2 are qualitatively very similar to the ones in Graph 1 as the same algorithms perform well for the same values of y. It is however important to realize that basically all results are more extreme. The performance of the largest (in/out-degree) first heuristic and the largest in-degree first heuristic is even worse than in Graph 1 and the performance of the other non-linear-program procedures is even better than in Graph 1, while their decrease in performance as y goes from one to three is even more pronounced. The highest sample standard deviation was again the sample standard deviation of the 5000 differences between the optimal solution makespan and the makespan of the solution of the largest (in-/out-degree) heuristic. This time this sample standard deviation was 0.52. The computation of the linear programming solution dropped significantly as y increases (from 2332/5000 for $y = 1$ to 19/5000 for $y = 25$), again the largest percentage drops occurred for the smaller values of y.

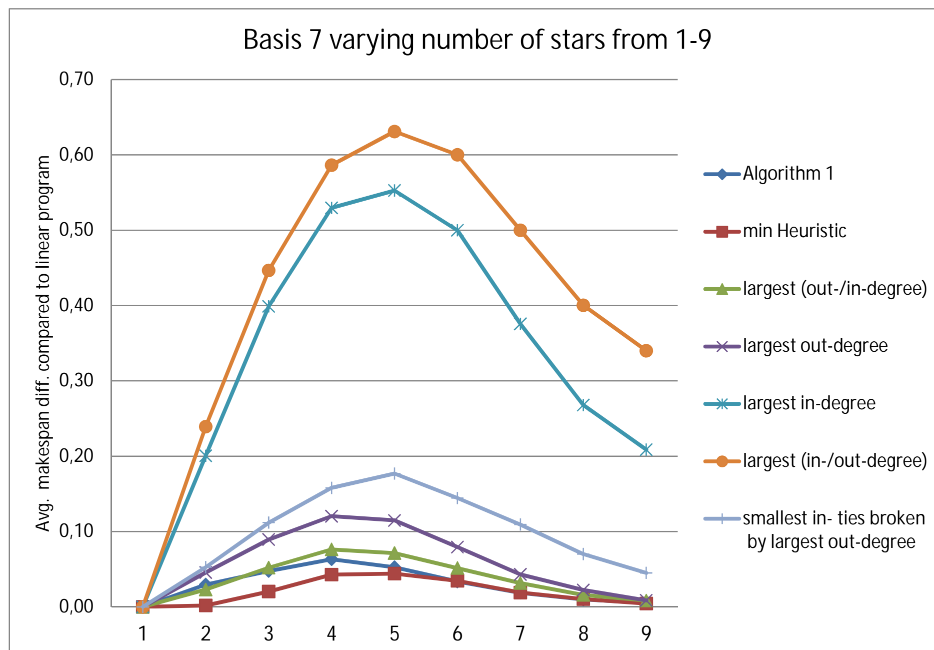
4.2.2 Varying the number of stars

When we vary the number of stars k, while keeping the other parameters constant, we get the following results.



Graph 3: Results for Basis 3 when varying the number of stars

Again the Heuristic 3 and 4 perform worst, but the most striking feature of Graph 3 is clearly that all non-linear-programming algorithms have their worst performance for $k = 3$. If the ratio of number of stars to number of machines is either small or large all procedures perform quite well. However for a certain middle range of the ratio all procedures perform quite badly, although that mid range does not seem to be independent of the values of k and m as we will see in the following graphs. This result is another main result in this paper. The highest sample standard deviation is again measured for the largest (in-/out-degree) heuristic with 0.49 and again the sample standard deviation follows the sample mean of the performance results for the different procedures. The number of times that the linear programming solution was computed dropped extremely as k was increased (from 3739/5000 for $k = 1$ to 3/5000 for $k = 5$). However, interestingly the largest percentage drops occurred for the larger values of k.

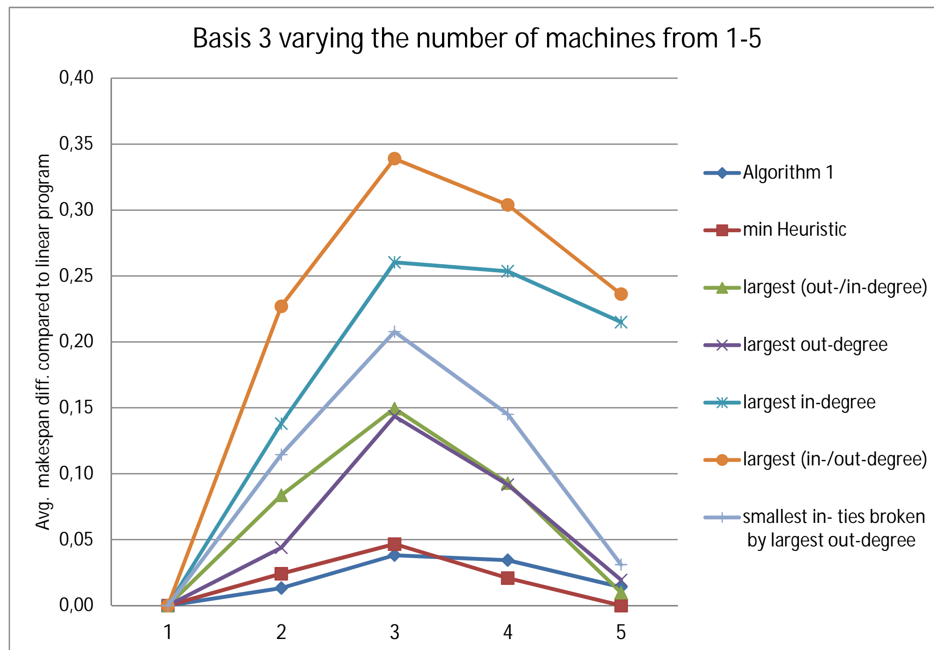


Graph 4: Results for Basis 7 when varying the number of stars

Similarly to what we found when varying y, the results for varying k get more extreme when we vary around Basis 7 instead of Basis 3. Especially the performance of Heuristic 3 and 4 is a lot worse than in the Basis 3 case. The performance of the other non-linear programming procedures seems to be similar to their performance in the Basis 3 case, although the values for which they underperform are shifted. Again the most striking feature of Graph 4 is probably the fact that all non-linear programming procedures perform badly around $k = 4$ and $k = 5$ although they do not all perform worst for the same value of k, as they did in the Basis 3 scenario. Also note that while in the Basis 3 scenario the procedures performed worst for $k = m$ in the Basis 7 scenario they perform worst for $k < m$. It is worth mentioning that for $k = 2$ Algorithm 1 performs not only worse than the minimum over all heuristics, but even worse than a single heuristic, the largest (out-/in-degree) heuristic. For the results presented in Graph 4 the highest sample standard deviation is again measured for the largest (in-/out-degree) heuristic with 0.53 and the sample standard deviation follows the sample mean of the performance results for the different procedures. The number of times that the linear programming solution was computed dropped extremely as k was increased (from 4352/5000 for $k = 1$ to 14/5000 for $k = 9$). Similarly to the results for Basis 3, the largest percentage drops occurred for the larger values of k.

4.2.3 Varying the number of machines

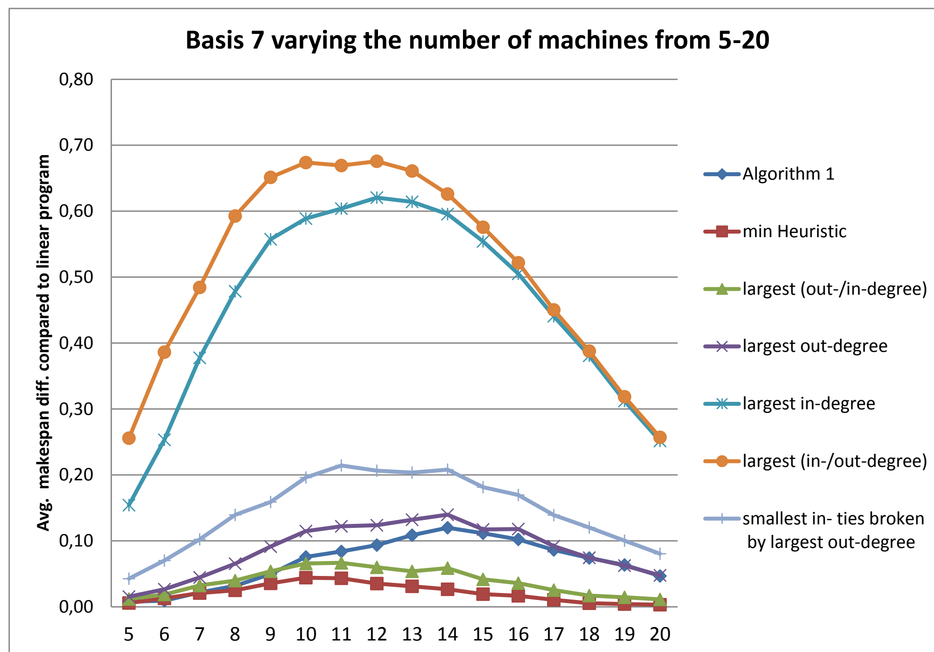
We will now have a look at what happens if we vary the number of machines m.



Graph 5: Results for Basis 3 when varying the number of machines

The results in Graph 5 look extremely similar to the result in Graph 3 although we are now varying a different parameter. In fact they look even more like the results of Graph 3 mirrored around the value 3, since the largest (out-/in-degree) heuristic performs better than largest out-degree heuristic and the minimum over all heuristics performs better than Algorithm 1 for $m = 2$ in Graph 5, while the same is true for $k = 4$ in Graph 3. Also, Algorithm 1 performs better than the minimum over all heuristics for $m = 4$ in Graph 5 while the same is true for $k = 2$ in Graph 3. This again supports our theory that the ratio of m and k in an instance is a good first indicator of the non-linear-programming procedures performance. As before, the highest sample standard deviation is measured for the largest (in-/out-degree) heuristic with 0.49 and the sample standard deviation follows the sample mean of the performance results for the different procedures. The number of times the linear programming solution was computed, thus the number of times that none of the non-linear-programming procedures found a makespan as solution that was equal to the makespan that one could achieve if precedence constrains

did not exist, increased strongly as m was increased (from 0/5000 for $m = 1, 2$ to 3160/5000 for $m = 5$). The fact that the number of linear programming solutions computed is equal to zero for $m = 1$ also makes intuitively sense, since if there is only one machine, precedence constraints can be fulfilled without additional cost since all jobs have to be scheduled in a different time slot anyway.



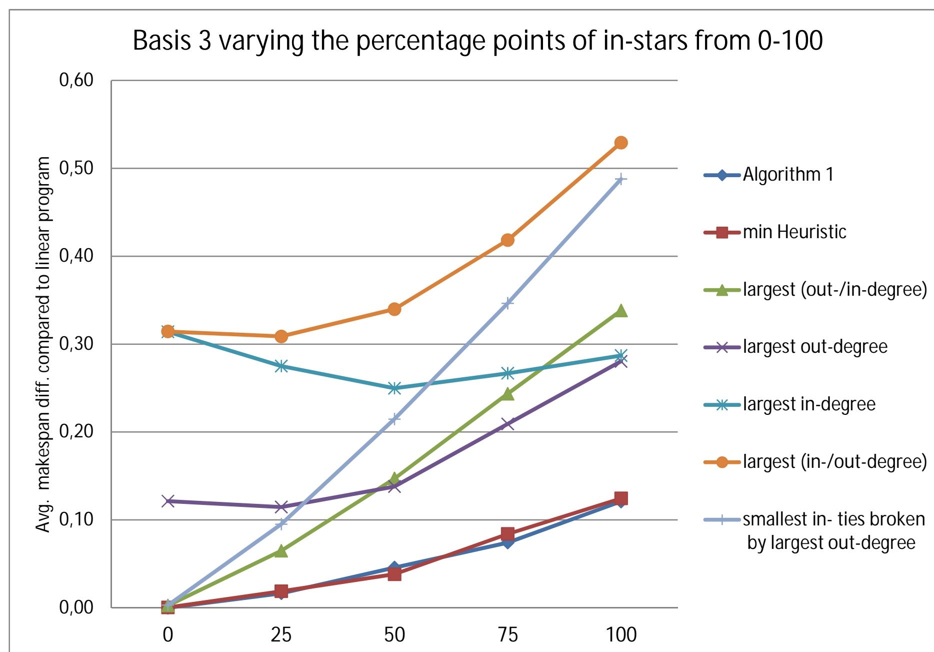
Graph 6: Results for Basis 7 when varying the number of machines

The results from Graph 6 are very similar to the results from Graph 5. They also seem to be a mirrored version of the results in Graph 4. What is special about these results however is that Algorithm 1 performs worse than the largest (out-/in-degree) heuristic for such a large section of values of m. We already expected that this might be the case for some values of m, since it was the case for one value of k in Graph 4. However the extent to which Algorithm 1 is outperformed by the largest (out-/in-degree) heuristic for larger values of m did surprise us. Another aspect that is special about this graph is that there does not seem to be a clear value of m for which all algorithms performed bad, instead all procedures seem to perform relatively bad for a whole range of values of m starting around $m = 10$ and going up to $m = 14$. As before, the highest sample standard deviation is measured for the largest (in-/out-degree) heuristic with 0.54 and the sample standard deviation follows the sample mean of the performance results for the

different procedures. Similarly to the results that we got from the computation of Graph 5, the number of times the linear programming procedure was called increased strongly with m (from 9/5000 for $m = 5$ to 4098/5000 for $m = 20$) with the largest percentage increases happening for the smaller values of m.

4.2.4 Varying the percentage points of in-stars

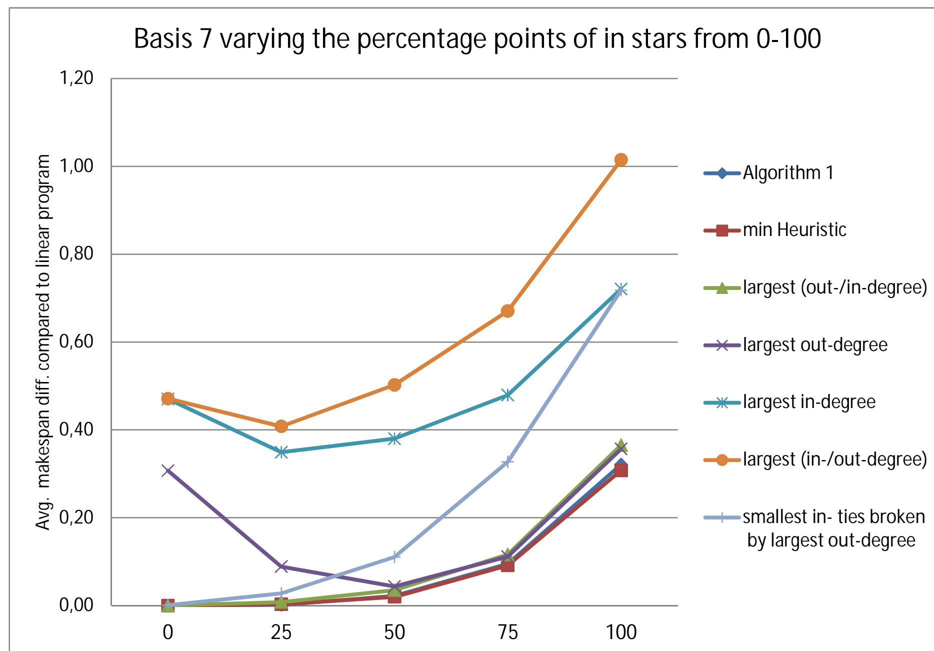
Lastly we consider what happens if we change the probability of creating an in-star $p/100$.



Graph 7: Results for Basis 3 when varying the percentage of in-stars

We were quite surprised by the results in Graph 7, since all algorithms except for the largest in-degree heuristic seem to be biased towards scheduling instances with less in- than out-stars well. We would like to note that this implies, that when using any of the other heuristics to solve an instance that has a lot more in-stars than out-stars it might be possible to improve the result by applying the following procedure. First convert every in-star in its mirror out-star and every out-star in its mirror in-star. Then solve the instance and finally mirror the solution schedule found around its middle time slot in order to find a solution to the original problem. Since both mirroring processes can be done in $O(k)$ time, doing this will not increase worst case complexity of the

procedures. It is also worth noting that in this graph for $p = 100$ we find the first setting in which the largest in-degree heuristic actually beats the largest (out-/in-degree) heuristic. Unsurprisingly, the highest sample standard deviation is still measured for the largest (in-/out-degree) heuristic with 0.54 and the sample standard deviation follows the sample mean of the performance results for the different procedures. The number of times the linear programming procedure was called slightly increased as p increased (from 250/5000 for $p = 0$ to 428/5000 for $p = 100$).



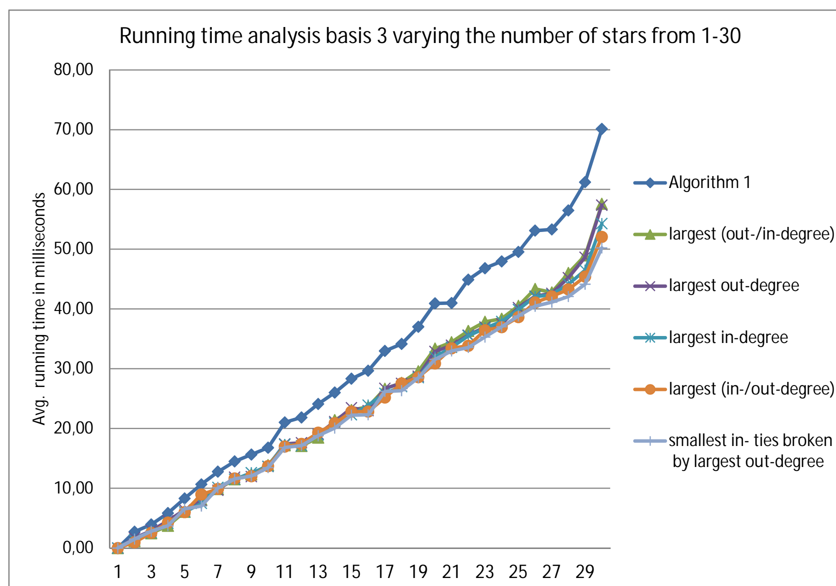
Graph 8: Results for Basis 7 when varying the percentage of in-stars

While the results in Graph 8 are relatively similar to the results in Graph 7 there are still some important differences. The first one is that heuristics 3-5 are performing a lot worse than before. For $p = 100$ Heuristic 4 even performs on average slightly more than a whole time-slot worse than the optimal solution, which strikes us as a surprisingly bad performance since it implies that the makespan Heuristic 4 found was more often by 2 worse than the optimal makespan than it was equal to the optimal makespan. Something else that surprised us was that for Basis 7 the performance of the largest out-degree heuristic is almost completely symmetric around $p = 50$ which was not at all the case for the Basis 3 performance. Surprisingly the highest sample standard deviation is measured for the smallest in- ties broken by largest out-degree heuristic with 0.62. Similarly interesting, the highest sample standard deviation

for the largest (in/out-degree) heuristic did occurs for $p = 75$ and not for $p = 100$ although the largest (in-/out-degree) heuristic performs much worse for p=100, breaking the pattern of strong linkage between sample standard deviation and sample mean. This implies that that the largest (in-/out-degree) heuristic seems to be quite consistently by one time slot worse than the optimal solution for $p = 100$. Slightly different from the results for Basis 3 the number of times the linear programming procedure was called strongly increased as p increased (from 17/5000 for $p = 0$ to 1151/5000 for $p = 100$).

4.3 Running time analysis

In this section we will present our results from testing how fast the different non-linear programming algorithms get a solution, when we take the Basis 3 model and vary the number of stars k from 1-30. Note that the results in this section are for several reasons less precise than the results in the earlier sections. This is because computation time is significantly influenced by things as the computer used and the data-structure used in the implementation of the algorithms and the precise way in which the algorithms are implemented. Thus only take the results in this section as a qualitative indicator instead of taking them as a precise quantitative assessment.



Graph 9: Running time for Basis 3 when the number of stars

As can be seen in Graph 9 the running time of all the heuristics is almost identical for all

values of k while the running time of Algorithm 1 is a bit slower than those of the other heuristics with a widening gap as the number of stars scheduled increases. Overall these results are quite close to our expectations of what the running time would look like, although we expected the gap between Algorithm 1 and the heuristics to widen faster than it did. We think that this slow widening of the gap might be due to the sorting part of both types of procedures having a higher overhead than the rest of the procedure. This would lead to a strong influence of the speed of the sorting part of the procedures on the overall speed of the procedures. Since the sorting algorithm that we used for all non-linear programming procedures is a merge sort algorithm with a different sorting condition for each procedure this might explain why the total running time is relatively similar for Algorithm 1 and the heuristics.

5 Conclusion

We have four results that we found particularly interesting. Our first interesting result is that there exist instances where Heuristic 1,3,4 and 5 performed very badly in the sense that their solution was by two time slots worse than the optimal solution. We did not find a similar result for Heuristic 2 although we checked every instance that we generated for the results in Section 4.2. Our second interesting result is that Algorithm 1 does not always find an optimal solution. Even more generally that no procedure that either always schedules all pure centers at the beginning or always schedules all pure centers at the end will always find an optimal solution for in-and out-star scheduling problems. Neither of this happened if we restricted ourselves to instances where both the in- and the out-degree of every in- and out-star was at least one. Our third interesting result is that all non-linear-programming procedures seem to perform badly if the number of stars is close to but below the number of machines in the instance. Since the maximal height of the precedence graph that we consider is two, thus the longest chain of related jobs is three, we suspect that we find this result because the non-linear-programming procedures might find deciding if an the optimal makespan for an instance is in three, four or five more difficult than the comparison of other candidate makespans. We found this thought particularly interesting since Berger et al. (2014) proves that for precedence constrains forming general stars deciding whether the optimal makespan is at most four is NP-complete if the measure of input size is proportional to the number of stars k. Note that these conclusions from our third interesting result are relatively speculative. Our fourth interesting result is that there exists no procedure or combinations of procedures that greedily schedules centers as early as possible, which always finds an optimal solution for all instances of the in- and out-star scheduling problem. Other results of our work are, that in general Heuristic 3 and 4 do perform worst for the instances we tested while Heuristic 1 and 2 perform best. Also most procedures seem to perform best if

the number of in-stars is lower than the number of out-stars. The minimum over all heuristics performs similarly to Algorithm 1, unless the number of machines m is large compared to the other parameters. Finally the running time of Algorithm 1 increases faster than the running time of the heuristics as k increases.

References

Berger, A., Grigoriev, A., Heggernes, P., and Z., R. (2014). Scheduling unit-length jobs with precedence constraints of small height. *OPERATIONS RESEARCH LETTERS*, 42(2):166–172.

Dolev, D. and Warmuth, M. K. (1984). Scheduling precedence graphs of bounded height. *Journal of Algorithms*, 5(1):48 – 59.

Gabow, H. N. (1981). A linear-time recognition algorithm for interval dags. *Information Processing Letters*, 12(1):20 – 22.

Hu, T. C. (1961). Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848.

Papadimitriou, C. H. and Yannakakis, M. (1979). Scheduling interval-ordered tasks. *SIAM Journal on Computing*, 8(3):405–409.

Ullman, J. (1975). Np-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393.