Distributing Flexibility to Enhance Robustness in Task Scheduling Problems

Daan Wilmer Tomas Klos Michel Wilson

Algorithmics Group, Delft University of Technology

Abstract

Temporal scheduling problems occur naturally in many diverse application domains such as manufacturing, transportation, health and education. A scheduling problem arises if we have a set of temporal events (or variables) and some constraints on those events, and we have to find a schedule, which is an assignment of values to the variables that satisfies the constraints. The execution of schedules in practice is typically surrounded by uncertainty, so that it makes sense to assign intervals rather than fixed times to events. Such a schedule is hypothesized to be more robust to disruptions, as it leaves room for adapting the assignment of exact times to events, to disturbances occurring during execution.

In previous work, we have shown how to efficiently compute an assignment of intervals to the variables in a temporal scheduling problem, that maximizes the sum of the lengths of the intervals. We empirically evaluated whether we can further improve the robustness of such a schedule by changing the distribution of intervals. In the current paper, we investigate in more detail how characteristics of the input instances affect different scheduling methods' robustness properties. From this investigation, we derive three new methods for designing interval schedules, and show them to provide similar or improved robustness.

1 Introduction

Scheduling problems are very common in many different application domains. Informally, we speak of a temporal scheduling problem when we have a number of temporal variables and some constraints on the values we can assign to them, and we are asked to construct a schedule, which is an assignment of values to the variables that satisfies all constraints. In this paper the events of interest are the starting times for tasks to be executed by agents. The constraints impose a precedence relation on the tasks: Some tasks can only start after certain other tasks have finished.

Very often, while a schedule is created, there is uncertainty about the environment in which it will be executed: tasks may turn out to take longer than anticipated, machines may break down, etc. Then it makes sense to assign to each task not an exact, fixed time, but rather an *interval* in which the task can start. That way, a precise starting time can be chosen from a task's interval depending on the circumstances unfolding during schedule execution, such as delays of preceding tasks. We interpret the length of the interval assigned to a task as the *flexibility of the task*. It seems intuitively attractive to construct an assignment of starting time intervals to tasks that maximizes the sum of the interval lengths we assign. This sum is then interpreted as the *flexibility of the schedule*.

In previous work [4], we showed that an interval schedule that maximizes flexibility can be computed in polynomial time. We can reasonably expect such a maximally flexible interval schedule to maximize the schedule's robustness in the face of disturbances occurring during executions, and to *minimize the tardiness* due to delays. However, sometimes it is not just the project itself, but the individual activities that must be finished in time—for example because of contracts with other parties. In this case we should aim to *minimize the number of delayed tasks*. In [12] we studied the effects on these two performance indicators

of three alternative heuristics for distributing flexibility, compared to the flexibility maximizing distribution (see Section 2.3).

In the current paper we investigate those results more thoroughly. In particular, we compute several characteristics of the input instances used in [12], and correlate those with the performance of the various methods used to distribute flexibility (Section 3). We also investigate the actual distributions of flexibility generated by the different heuristics. Based on these results, we formulate two explanatory hypotheses regarding the differential performance of these heuristics, which we try to refute in a further series of experiments (Section 4). These inspire the design of several new heuristics, which our experiments show to produce very robust schedules (Section 4).

2 Preliminaries and Background

In this section, we introduce some notation describing our framework, and summarize the problem and solution methods from [12]. The framework we use is that of the task scheduling problem [9]. An instance of this problem consists of a set $T = \{t_1, \ldots, t_n\}$ of n tasks, together with a *precedence relation* $\prec \subseteq T \times T$. If $(t_i, t_j) \in \prec$, we say t_i *precedes* t_j, which means t_i has to be completed before t_j can start. We say that t_i *immediately precedes* t_j (written as $t_i \ll t_j$), if $t_i \prec t_j$ and there does not exist a $t_k \neq t_i, t_j$ such that $t_i \prec t_k$ and $t_k \prec t_j$. Each task t_i has a *processing time* $p_i \in \mathbb{R}^{\geq 0}$, a *release date* $r_i \in \mathbb{R}^{\geq 0}$ and a *due date* $d_i \in \mathbb{R}^{\geq 0}$. A task cannot start before its release date, and it must be completed at or before its due date. We usually refer to an instance of the task scheduling problem as a tuple $S = (T, C)$, where C is the set of constraints imposed by the relation $\prec$ and by the tasks' release and due dates.

The task scheduling problem is to find a *schedule* $\sigma : T \to \mathbb{R}^{\geq 0}$, which is an assignment of start times to tasks that satisfies all constraints in C, encoding precedence relations and release and due dates. Deciding whether such a schedule exists and if so, finding a schedule for a task scheduling problem can be done in $O(n+m)$-time, where m is the number of precedence tuples [9]. The earliest and latest starting times $est(t)$ and $lst(t)$ of all tasks t can also be found in $O(n + m)$-time.

2.1 Flexibility in Scheduling

Intuitively, the flexibility of a task scheduling system S is related to the number of schedules that exist for S: The more different schedules an instance allows, the more *flexible* we would say it is. Since determining the exact number of different schedules is infeasible, different *flexibility metrics* have been proposed to capture this notion [1, 2, 3, 6]. Upon close inspection, existing flexibility metrics turn out to overestimate the amount of flexibility inherent in task scheduling instances, due to their inability to take into account *dependencies* between tasks (see [11, 12] for detailed discussions).

For example, the difference $lst(t_i) - est(t_i)$ is in principle an appropriate way to measure the flexibility of a single task t_i, in the sense that it accurately reflects the freedom we have in shifting the starting time of task t_i between $est(t_i)$ and $lst(t_i)$. However, if task t_i is precedence-related to task t_j, then the sum of these individual flexibilities will typically overestimate the total freedom we have in assigning times to tasks t_i and t_j. The reason is that not every unit of flexibility for t_j may be avialable once t_i has been assigned a starting time. Suppose two tasks t_i and t_j have release times of 0, due dates of 5, and processing times of 1, while the precedence constraint $t_i \prec t_j$ holds. Then $est(t_i) = 0$, $est(t_j) = 1$, $lst(t_i) = 3$, and $lst(t_j) = 4$. For both t_i and t_j, then, $lst - est = 3$, but the total freedom in this instance is *not* $3 + 3 = 6$: Once the starting time for t_i is fixed at any value > 0, the freedom for t_j is decreased.

We propose a flexibility metric that measures the flexibility of a task as the size of an interval as before, but we construct *independent* intervals, in which dependencies between tasks have been accounted for. We call such a set of intervals an *interval schedule*.

Definition 1 (Interval schedule)**.** *Given a task scheduling instance $S = (T, C)$, an* interval schedule for *S is a function $I_T : T \to \{[\ell, u] \mid \ell, u \in \mathbb{R}^{\geq 0}, \ell \leq u\}$, such that for every tuple $(v_1, v_2, \ldots v_n)$, where $v_t \in I_T(t) = [a_t, b_t]$ for all t, the function σ defined by $\sigma(t) = v_t$ for all t, is a schedule for S.*

Note that a schedule satisfies all constraints by definition.

(We sometimes write an interval schedule simply as a set of intervals, making it obvious from the written ordering of the elements of the set which interval belongs to which task.) So an interval schedule assigns (independent) intervals to tasks in such a way that, for *every* task, we can pick *any* value in its interval as an element of a fixed-time schedule for S—*irrespective* of the values chosen for other tasks. We can now measure the flexibility of a task as the length of the interval assigned to it in an interval schedule.

Definition 2 (Flexibility of a task)**.** *Given a task scheduling instance $S = (T, C)$ and an interval schedule $I_T = \{[a_t, b_t]\}_{t \in T}$ for S, the* flexibility of task t in interval schedule I_T *is defined as $\mathit{flex}(t, I_T) = (b_t - a_t)$.*

Next we define the flexibility of a schedule for S.

Definition 3 (Flexibility of a schedule)**.** *Given a task scheduling instance $S = (T, C)$ and an interval schedule $I_T = \{[a_t, b_t]\}_{t \in T}$ for S, the* flexibility of schedule I_T *is defined as $\mathit{flex}(S, I_T) = \sum_{t \in T} \mathit{flex}(t, I_T)$.*

There may exist several different interval schedules for a given system S, each with possibly different flexibility values. In the earlier example, we could assign $[0, 1]$ to t_i and $[2, 3]$ to t_j, or $[0, 1]$ to t_i and $[2, 4]$ to t_j: in both cases, any combination of values from the two intervals constitutes a schedule, but these two interval schedules incorporate different flexibilities (2 and 3, respectively).

Definition 4 (Flexibility of an instance)**.** *Given a task scheduling instance $S = (T, C)$, the* flexibility of S *is defined as $\mathit{flex}(S) = \max_{I_T} \mathit{flex}(S, I_T)$.*

Of course we would like to compute the flexibility of an instance S efficiently. The following proposition offers a characterization of interval schedules that helps us in doing this.

Proposition 1. *Given a task scheduling instance $S = (T, C)$, a set of non-empty intervals $I_T = \{[a_t, b_t]\}_{t \in T}$ is an interval schedule for S if*

1. *for all $t \in T$, $est(t) \leq a_t \leq b_t \leq lst(t)$;*
2. *for all $t, t' \in T$, $t \prec t'$ implies $b_t + p_t \leq a_{t'}$.*

Proof. See [12]. □

Note that the conditions stated in Proposition 1 are all linear, and if we want to maximize flexibility, we have a linear function to maximize. Hence, we can efficiently find the (maximal) flexibility of a task scheduling system $S = (T, C)$ by solving an appropriately specified linear program (see [12] for details). The solution to this LP consists of values for the lower and upper bounds t_i^- and t_i^+, respectively, of the interval assigned to task t_i (for all i).

2.2 Distributing Flexibility

Although the solution proposed above yields an interval schedule that offers maximal flexibility, this may not be the most appropriate schedule in all circumstances. Moreover, it does not always yield a unique schedule. Consider the instance S_{seq} in Figure 1, consisting of five sequential tasks $t_a, \ldots, t_e$, where for all tasks i, $r_i = 0$, $d_i = 10$ and $p_i = 1$. The flexibility in this instance is $\mathit{flex}(S_{\text{seq}}) = 5$. Two in-

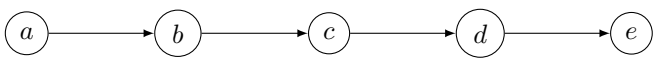

Figure 1: Task scheduling instance S_{seq} with five sequential tasks.

terval schedules achieving this flexibility score are $I_{T,r} = \{[0, 0], [1, 1], [2, 2], [3, 3], [4, 9]\}$ and $I_{T,\ell} = \{[0, 5], [6, 6], [7, 7], [8, 8], [9, 9]\}$, which place all flexibility at the rightmost and the leftmost task, respectively. Both schedules have the same flexibility, but $I_{T,\ell}$ is much more vulnerable if task processing times

are increased due to unexpected delays: For every task except t_a such a delay has a direct effect on the makespan of the schedule. In contrast, $I_{T,r}$ is much less vulnerable to delay, since all flexibility is placed at the end of the schedule. When violations of individual tasks' scheduled finish times are costly, for example because of agreements with other parties, minimizing the expected number of violations becomes important. Then, a schedule like $I_{T,e} = \{[0,1],[2,3],[4,5],[6,7],[8,9]\}$ would be more appropriate: It assigns some flexibility to *every* task, so delays are less likely to propagate to subsequent tasks.

While this example shows that it is unclear *which* maximally flexible interval schedule is the most appropriate, the next example shows that even the appropriateness of the objective of maximizing flexibility is debatable. The instance S_{par} in Figure 2 extends instance S_{seq} with a parallel component (at task b).

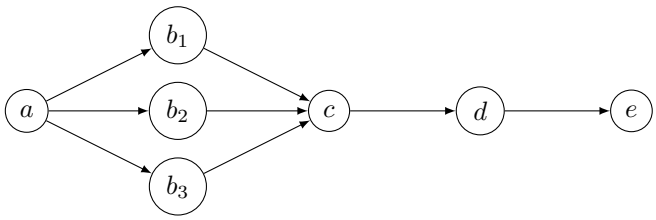

Figure 2: Task scheduling instance S_{par} with a parallel component.

Maximizing $\mathit{flex}(S_{\text{par}})$ yields the unique schedule $I_{T,m} = \{[0,0],[1,6],[1,6],[1,6],[7,7],[8,8],[9,9]\}$, with a flexibility of 15. Notice that all flexibility is concentrated in the parallel parts of the schedule, because assigning flexibility there gives a multiplicative effect, and that there is no flexibility in the sequential part of the schedule. Although it sacrifices some flexibility (8 units), a schedule like $I_{T,n} = \{[0,1],[2,3],[2,3],[2,3],[4,5],[6,7],[8,9]\}$, may be more robust in the face of execution-time delays: All tasks now have some flexibility to absorb delays, which may lead to a lower number of *violations*, defined as tasks starting outside their assigned intervals.

The following three alternative heuristics to assign flexibility are proposed in [12].

equalized This method finds values for t^- and t^+ (for all t) that minimize

$$\sum_{t \in T} ((\mathit{lst}(t) - \mathit{est}(t)) - (t^+ - t^-))^2, \tag{1}$$

which is like minimizing the variance: We want flexibility to be distributed as equally as possible. For the rationale, consider again the example in Section 2.1, where the tasks t_i and t_j both have an $[\mathit{est}(t), \mathit{lst}(t)]$ interval of length 3. If we assign to t_i an interval of length 3, then t_j receives an interval of length 0, and the objective function above has value 9. However, if we equalize the assignment by assigning to both tasks an interval of length 1.5, the objective function is minimized (at 4.5).

weighted, all Because delays can be expected to differentially affect tasks at different locations in the task graph, this method and the next employ a weighting factor, and assign intervals that minimize

$$\sum_{t_i \in T} w(t_i) \cdot ((\mathit{lst}(t_i) - \mathit{est}(t_i)) - (t_i^+ - t_i^-))^2$$

where $w : T \to \mathbb{R}$ is a weight function influencing the amount of flexibility t_i receives. This method ('weighted, all') uses a task's *total number of predecessors* as its weight, with the rationale that in each predecessor of t_i, a delay can occur which may eventually impact t_i. The more predecessors a task has, the more at risk it is of being impacted by such a delay.

weighted, direct If delays are relatively small, and each task has some flexibility, it is reasonable to assume that tasks which are distant from t_i (in terms of the path length) have less impact than tasks which are close. With the same weighted objective function as the previous method, this method only considers the *number of immediate predecessors* of a task as its weight.

2.3 Evaluation

The methods proposed in [12] were evaluated in simulations of schedule execution environments. The 120-task instances of the PSPLIB benchmark set [7] were used, whereby all resource constraints were ignored. A due date was imposed of 1.1 times the makespan of the earliest start time schedule of each instance. For each instance, interval schedules were first constructed using each of the four methods (maximized flexibility, equalized, weighted-all and weighted-direct). (Since the maximal flexibility objective may return different solutions, we pick the one that minimizes Eq. (1).) These schedules were then executed in an environment in which random increases in processing time were introduced for a randomly chosen subset of the tasks. The number of delayed tasks was varied between 12 (10%) and 60, in steps of 12. The amount of delay was varied in percentages of delayed tasks' processing times, between 10% and 100% (in steps of 10%), and between 100% and 200% (in steps of 25%). The *makespan* was measured (as well as the *tardiness* compared to the makespan of the undelayed setting), as well as the *number of violations*, defined as a task starting outside its assigned interval.

The results showed among others that, as expected, 'dispersing' flexibility rather than maximizing it comes at the expense of flexibility: Averaged across all experiments, about 5% flexibility is lost when equalizing it, increasing to about 29% when taking all predecessors into account: Maximizing flexibility requires assigning flexibility to parallel components (see Figure 2), while weighting by number of predecessors assigns flexibility to tasks where parallel components join together. The *number of violations* drops from 46 (with the maximized flexibility distribution) to 39 when using an equalized flexibility distribution, and increases from 44 to 54 when taking direct or all predecessors into account: As explained above, too little flexibility is assigned to absorb delays in this case. On the other hand, weighting tasks by their number of predecessors when assigning flexibility has a beneficial effect on *tardiness*, since more propagated delays can be absorbed by tasks with large intervals occurring at the end of the schedule.

We reproduce Figures 4 and 5 from [12] as our Figure 3, and we refer the reader to [12] for more results.

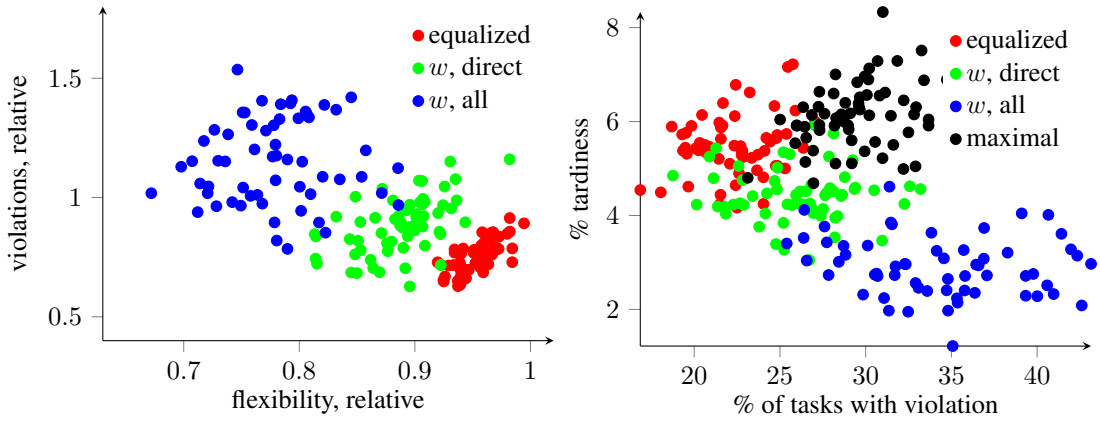

Figure 3: Figures 4 and 5 from [12]: 60 random instances.

While these figures generally support the summaries given above (such as the ordering of methods by their loss of flexibility relative to the distribution with maximized flexibility, and by the number of violations they incur), they clearly indicate that there is quite some variance across different instances. Moreover, in the figure on the left, there seems to be a correlation between the loss of flexibility and the number of violations, most clearly for the equalized distribution, that can not be explained by these numbers alone. Because we think results like these are due to problems instances' specific task networks, we focus on this aspect to give a more complete explanation of the overall results, and build an 'empirical theory' [5] of this algorithm.

3 Exploratory Study

In particular, we address the following questions in this paper.

1. How do properties of instances affect the number of violations they suffer?
2. How is performance determined by the properties of the resulting flexibility distribution?

Instance properties As to the first question, we computed several characteristics of instances, and correlated them with the number of violations. The results of the full analysis are in [10]; here, we will focus on the strongest correlation we observed, between the number of violations and a metric we call an instance's *complexity*, which we measured as the 'cyclomatic complexity' [8] of the instance's constraint graph. This is originally a measure of a program's complexity, measuring the number of linearly independent paths through a program's source code.[1] Translated to our context, it is computed as the difference between the number of constraints and the number of tasks. Since all instances have the 120 tasks, higher complexity means more constraints, so tasks have on average more predecessors and successors.

An explanation might thus be that if a task is delayed by more than its flexibility, the 'excess delay' propagates to its successors, so if it has on average more successors, this leads to more violations. Another explanation, taken from [12], could be that tasks that have more predecessors are more likely to have at least one predecessor having 'excess delay.' Since graphs with a higher complexity have more predecessors per task, the probability that a task receives propagated delay is higher, thereby increasing the number of violations. However, an algorithm based on this reasoning already failed to even match the performance of a similar algorithm that does not take this reasoning into account [12]. It is therefore less likely that this explanation is correct.

Flexibility distribution For the second question, we looked at the distributions of flexibility generated by the various methods. Figure 4 illustrates these distributions (of the maximized and the equalized methods)

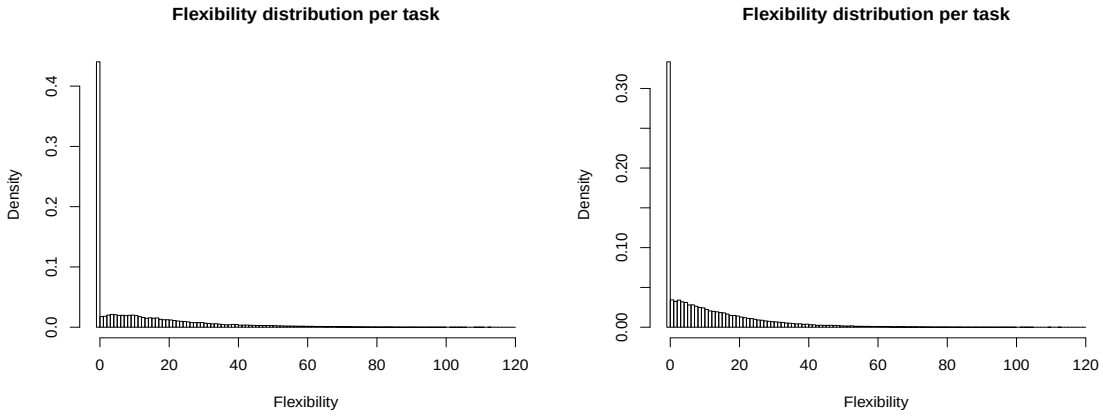

Figure 4: Flexibility histogram for the maximized (left) and equalized (right) distributions.

with histograms that show the proportion ('density') of tasks with a flexibility in the bins shown, averaged over all instances. The bins are chosen as $[-1, 0], (0, 1], (1, 2], \ldots$, so that the first bin only counts tasks with a flexibility of zero, as negative values for flexibility are not possible. Apparently, even the method we call 'equalized' is unable to reduce the large number of tasks in the first bin significantly. (Note that the scales on the y-axes are not the same.) Apart from the number of zero-flexibility tasks, we looked at the correlations

[1] `http://en.wikipedia.org/wiki/Cyclomatic_complexity`

with two other instance characteristics, namely the sum over all tasks of a task's flexibility multiplied with the number of predecessors and the number of successors, respectively. These numbers measure to what extent the flexibility is concentrated in tasks that have many predecessors and successors, respectively.

Again, the full correlation results can be found in [10]. To summarize, the correlation between the number of zero-flexibility tasks and number of violations is strongest in settings with many small delays, and weakest when there are many large delays. We hypothesize that this is because when a task has no flexibility at all, any delay it gets will be propagated to its successors. Thus, when more tasks have no flexibility, delays will be propagated more, resulting in more violations. However, when the total amount of delay becomes too high, this effect might be negated because tasks get more delay than can be handled by flexibility. In this case the number of zeros makes little difference: delays will be propagated and violations will be plentiful anyway. The second observation that can be made is that the flexibility multiplied by the number of predecessors is not correlated with the number of violations. This matches the results in [12], and refutes the second explanation mentioned above for the correlation between complexity and number of violations. The third observation is that the flexibility multiplied by the number of successors has a negative correlation with the number of violations. This supports the first explanation above for the correlation between complexity and number of violations.

4 Hypotheses and Experiments

Based on this exploratory study, we propose two hypotheses about the heuristics' behavior.

1. Variations in algorithm performance (in terms of number of violations) across instances are caused by instances' complexity, due to the fact that in instances with higher complexity tasks have more successors so that delays are propagated to more tasks, which increases the number of violations.

2. Differences between heuristics are caused by two factors, the first of which is more significant when delays are relatively small, while the second is more significant when delays are relatively large.

   (a) The number of tasks that have no flexibility, because when there are more tasks without flexibility delays will be propagated more, causing more violations;

   (b) The assignment of flexibility to tasks that have many successors, because when a task has more successors, delays will be propagated to more tasks, causing more violations.

To test the first hypothesis, experiments could be run in which the complexity of the instances is controlled. All other parameters being equal, this should result in lower variance in the number of violations. Results for experiments testing these predictions are reported in [10], but because of space limitations we don't discuss them here any further than to say that our experiments confirm this hypothesis.

The second hypothesis can be tested by experimenting with new methods for distributing flexibility. If the *first* factor indeed plays a role, a distribution that has less tasks with zero flexibility should perform better. The method 'max_minflex' calculates a minimum flexibility $f_{\min}$ to assign to each task, and maximize this value such that the schedule still finishes before the deadline. Then, flexibility is maximized under this additional constraint of assigning $f_{\min}$ to each task, and is distributed as equalized as possible. If the *second* factor is correct, then with a heuristic that assigns more flexibility to tasks with many successors, less delays will be propagated to those successors, which will decrease the number of violations. The method 'wsucc' uses a task's number of successors as its weight in the weighted objective function from above. Finally, a combination ('wsucc_minflex') of these two methods gives $f_{\min}$ to each task, and then uses the weighted distribution based on number of successors to divide any remaining flexibility.

We used two settings for the delay parameters: 80% of tasks delayed by 5% and 80% of tasks delayed by 80%, because these gave the strongest correlations in previous experiments. The results are shown in Table 1. When delays are small, tasks are never delayed more than the flexibility they are assigned by max_minflex can compensate. This makes sense, in light of the fact that the deadline is set at 1.1 times the makespan,

Flexibility distribution	80% delayed by 5% #violations	80% delayed by 5% outperforms equalized	80% delayed by 80% #violations	80% delayed by 80% outperforms equalized
equalized	31.66	n.a.	70.88	n.a.
wsucc	35.67	22%	31.66	100%
max_minflex	0	100%	70.63	60%
wsucc_minflex	0	100%	66.17	91%

Table 1: Averages over all instances, comparing the equalized distribution and the new distributions.

and delays are never more than 5% of tasks' lenghts. With larger delays, max_minflex gives about the same number of violations as the equalized heuristic.

The wsucc heuristic that assigns more flexibility to tasks with many successors, shows the opposite effect: when there are large delays, it outperforms the equalized heuristic in all cases. When delays are smaller, it performs mostly worse than the equalized heuristic.

The results from wsucc_minflex, that combines the other two, also combine the results of the other two. When delays are small, it leads to no violations (like max_minflex); when delays are large, it outperforms the equalized algorithm in most cases (like wsucc), but not in all of them like the wsucc distribution.

Overall, we conclude that these experiments support the second hypothesis.

5 Conclusion and Future Work

We investigated the results by Wilson et al. [12] in detail. Based on an in-depth exploratory study, we formulated two hypotheses and performed a series of experiments to seek to refute their predictions. We found support for our hypotheses, that the complexity of an instance is influential in algorithms' performance in terms of the number of violations their schedules produce, and that algorithms can be improved if they assign a non-zero minimum flexibility to each task, and give more flexibility to tasks with many successors.

In future work, we would like to investigate the influence of other characteristics of instances and flexibility distributions. This may lead to improved ways of assigning flexibility, like making it sensitive to tasks' lengths, and to prior knowledge of the way delays are distributed in the network.

References

[1] M.A. Aloulou and M.C. Portmann. An efficient proactive-reactive scheduling approach to hedge against shop floor disturbances. In *Multidisciplinary Scheduling: Theory and Applications*, 2005.

[2] A. Cesta, A. Oddi, and S.F. Smith. Profile-based algorithms to solve multiple capacitated metric scheduling problems. In *Proceedings of the Artificial Intelligence Planning Systems*, 1998.

[3] H. Chtourou and M. Haouari. A two-stage-priority-rule-based algorithm for robust resource-constrained project scheduling. *Computers & Industrial Engineering*, 55, 2008.

[4] L. Endhoven, T. Klos, and C. Witteveen. Maximum flexibility and optimal decoupling in task scheduling problems. In *IAT*, 2012.

[5] J.N. Hooker. Needed: An empirical science of algorithms. *Operations Research*, 42, 1994.

[6] L. Hunsberger. Algorithms for a temporal decoupling problem in multi-agent planning. In *AAAI*, 2002.

[7] R. Kolisch, C. Schwindt, and A. Sprecher. Benchmark instances for project scheduling problems. In *Handbook on Recent Advances in Project Scheduling*, 1998.

[8] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2, 1976.

[9] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 2008.

[10] D. Wilmer. Investigation of "Enhancing flexibility and robustness in multi-agent task scheduling". arXiv, `http://arxiv.org/abs/1307.0024`, 2013.

[11] M. Wilson, T. Klos, C. Witteveen, and B. Huisman. Flexibility and decoupling in the simple temporal problem. In *IJCAI*, 2013.

[12] M. Wilson, C. Witteveen, T. Klos, and B. Huisman. Enhancing flexibility and robustness in multi-agent task scheduling. In *Proceedings OPTMAS*, 2013.