Efficient Workplan Management in Maintenance Tasks

Michel Wilson[a] Nico Roos[b] Bob Huisman[c] Cees Witteveen[a]

[a]*Delft University of Technology, Dept of Software Technology*
[b]*Maastricht University, Dept of Knowledge Engineering*
[c]*NedTrain, Fleet Services - Maintenance Development*

Abstract

NedTrain is a Dutch company tasked with performing the maintenance of the rolling stock of the national railway company, NS. NedTrain owns several workshops at different locations. The scheduling in one such workshop will be taken as point of departure for the discussion in this paper. After discussing a suitable representation of the NedTrain task workplan problem that allows for a significant speed up in answering basic questions about schedules, we address the problem of modeling possible delays of individual tasks in a NedTrain workplan in such a way that questions concerning the likelihood of violation of deadlines can be easily answered. In particular, a method is presented to make use of a probabilistic representation of the possible delays in task executions, making it possible to better evaluate the quality of the schedule with regard to makespan extensions and deadline violations.

1 Introduction

NedTrain is a Dutch company tasked with performing the maintenance of the rolling stock of the national railway company, NS. NedTrain owns several workshops at different locations. The scheduling in one such workshop will be taken as point of departure for the discussion in this paper.

Typically, each workshop is confronted with the maintenance of arriving rolling stock units that arrive at some time at the workshop and have to be repaired before an agreed upon departure time. For each unit a set of maintenance tasks has to be performed. The time window of each such maintenance task is limited by the arrival time and departure time of the rolling stock unit, the available resources (personnel availability, tool or track availability) and some precedence constraints that might be imposed upon the tasks themselves, such as requiring that a final inspection is the last task to be performed.

Taking into account all these restrictions, a preliminary work plan is composed where all the tasks belonging to the rolling stock units to be maintained are placed in a partial order. Based upon this workplan, NedTrain is able to answer basic questions as: what is the makespan of this schedule, i.e., what is the completion time of the last job scheduled in the workplan, and does the workplan satisfy all the deadlines imposed on the rolling stock units?

Example 1 *As a practical example, consider two rolling stock units. The first one arrives at 08:00, and on this train three maintenance tasks need to be performed, taking one, two and one hours respectively. The due time for this train is 19:00. The second train arrives at 9:00, and needs two tasks: one of two hours and one of three hours. This train has a due time of 22:00. Additionally, the last task of the first train shares a resource with the last task of the second train, and as such, these tasks cannot be executed in parallel.*

Often, however, disturbances of the proposed workplan occur. Tasks take more time than foreseen, additional tasks have to be scheduled because inspection has shown that additional repair actions have to be performed, or some tasks have to be delayed because of lack of personnel. Hence, NedTrain is not only interested in having a scheduling tool where scheduling repair actions can be performed quite easily and efficiently, but also in the question whether or not such scheduling repair actions are likely to be performed in the near future with the current workplan and if so, whether there exists a better schedule where such changes are less likely to happen.

In order to build a system capable of handling such questions, first of all, it is essential that basic operations on workplans such as computing schedules and checking whether or not deadlines will be violated can be done very efficiently. Currently, Nedtrain has built a prototype to solve such *task schedule problems* using Simple Temporal Networks (STNs, see [4]) as the underlying representation of tasks and their constraints. It is well-known ([3]) that solutions to such systems, i.e., suitable dispatching times of the jobs, can be found in polynomial time. Testing these methods in practice on representative scheduling problems originating from within NedTrain, it was found that a significant part of computation time was used purely creating and maintaining the temporal network representing the solution. Improving the computation time to manipulate operations on the STN could speed up the scheduling system significantly and would enable the representation and manipulation of larger scheduling problems than can be handled at this moment.

Therefore the first problem we will deal with in this paper is the following question:

> *Is there a suitable representation of the NedTrain task workplan problem that allows for a significant speed up in answering basic questions about properties of schedules derived from such work plans?*

As we remarked before, speeding up the current problem, however, is not the only problem NedTrain is interested in. In this paper we will also focus upon an important problem which is encountered at NedTrain, and probably in almost every other organization as well: the actual execution duration of jobs might often exceed the predicted execution time. Such delays can easily influence the rest of the schedule or the makespan, or they might even cause a violation of the deadlines imposed upon the completion of jobs. Therefore, we will also address the following problem:

> *Is it possible to model possible delays of individual tasks in a NedTrain workplan in such a way that questions concerning the likelihood of violation of deadlines can be easily answered?*

To address both these problems, we start by defining the underlying task scheduling problem and investigate its embedding in STNs. Then we show that common operations on task scheduling problems such as extracting solutions can be achieved much faster by using specialized algorithms rather than using STN-inspired solution methods. In particular we will show that common operations requiring $O(n^3)$ time on STNs can be performed in linear time using a direct representation.

Next, we will introduce delays of task durations and investigate methods to establish their consequences for makespan extensions as well as deadline violation. In particular, a method is presented to make use of a probabilistic representation of the possible delays in task executions, making it possible to better evaluate the quality of the schedule with regard to makespan extensions and deadline violations.

2 Preliminaries

With the global form of the problem outlined above, this section will be used to formalize the exact details. The first problem we have to address is the exact specification of the NedTrain maintenance scheduling problem. We call this problem the task schedule problem, where given a partial order of tasks to be executed we have to find out whether there exists a scheduling of the tasks meeting the deadlines specified.

2.1 The task schedule problem

We start by defining the task schedule problem.

Definition 1 *A* task schedule problem *is a five-tuple $(T, \prec, l, a, d)$, where T is the set of tasks to be scheduled, $\prec$ a precedence relation defined on T, $l : T \to \mathbb{N}^+$ defining the length of each task, $a : T \to \mathbb{N}^+$ the release (arrival) times, and $d : T \to \mathbb{N}^+$ the due times of the tasks. It is assumed that the precedence relation $\prec$ induces a partial order on the set of tasks T.*

The goal is to find assignments to each time point variable, satisfying the constraints imposed by the precedence relation, the task lengths, release times and due times.

Example 2 *The example problem described in the introduction can be modeled with five time points, one for each task. Points t_1, t_2 and t_3 correspond to the first train, and t_4 and t_5 to the second train. Time is measured in hours relative to a reference time point. If we take 08:00 as reference time point, $a(t_1) = a(t_2) = a(t_3) = 0$, since the first train arrives at 08:00. For the second train, we have $a(t_4) = a(t_5) = 1$.*

In similar fashion, the due times are set: $d(t_1) = d(t_2) = d(t_3) = 11$ *and* $d(t_4) = d(t_5) = 14$*. For the task lengths, we have* $l(t_1) = l(t_3) = 1$*,* $l(t_2) = l(t_4) = 2$ *and* $l(t_5) = 3$*. Lastly, the precedence constraints must be set. It is assumed that only one task can be performed at a time on each train, so we have* $t_1 \prec t_2 \prec t_3$ *and* $t_4 \prec t_5$*. Additionally, due to the resource constraint described between* t_3 *and* t_5*, either* $t_3 \prec t_5$ *or* $t_5 \prec t_3$ *has to be added. In this case,* $t_3 \prec t_5$ *is chosen.*

2.2 Finding solutions

To solve the task schedule problem, we need to find a start time for every task respecting all constraints. One way is to embed the problem in a so-called *Simple Temporal Network* or STN. Such an STN is a tuple $\mathcal{S} = (\mathcal{T}, \mathcal{C})$ where $\mathcal{T}$ is a set of time points and every constraint $c \in \mathcal{C}$ is of the form $\tau_i - \tau_j \leq w_{i,j}$ where $\tau_i, \tau_j \in \mathcal{T}$ and $w_{i,j}$ is a given constant. A solution of an STN $\mathcal{S}$ is an assignment σ of values to time points in $\mathcal{T}$ such that all constraints are satisfied. It is well-known that deciding whether or not such an STN has a solution or finding a solution can be done in polynomial time.

It is simple to embed a task schedule problem $(T, \prec, l, a, d)$ into a simple temporal network $\mathcal{S} = (\mathcal{T}, \mathcal{C})$:

- For each task $t_i \in T$, $\mathcal{T}$ contains two timepoints $\tau_{i,1}$ and $\tau_{i,2}$ representing the starting time and the end time, respectively, of task t_i. Besides these time points a reference time point z is added to $\mathcal{T}$.
- For each precedence constraint $t_i \prec t_j$, $\mathcal{C}$ contains two constraints $\tau_{j,1} - \tau_{i,2} \leq \infty$ and $\tau_{i,2} - \tau_{j,1} \leq 0$.
- For each task $t_i \in T$, $\mathcal{C}$ contains two constraints $\tau_{i,2} - \tau_{i,1} \leq l(t_i)$ and $\tau_{i,1} - \tau_{i,2} \leq -l(t_i)$ to encode the task durations.
- And, finally, for each task $t_i \in T$, $\mathcal{C}$ contains the constraints $z - \tau_{i,1} \leq -a(t_i)$ and $\tau_{i,2} - z \leq d(t_i)$ to encode the restrictions on release time and the due time.

To find a solution for an STN, the Floyd-Warshall algorithm can be used to specify the earliest and latest starting times. Finding such solutions requires roughly $O(n^3)$ time.

Although STNs might be used to solve the task schedule problem, we can solve quite a lot of problems much faster by specializing algorithms to task schedule problems instead of solving them via their STN representation. We will start by computing the minimal and maximal solutions of a task schedule problem and then show how arbitrary solutions can be obtained in linear time instead of cubic time.

Such a more efficient computation of the earliest and latest start times is possible by exploiting the topological ordering defined by the precedence relationships. Defining $\text{PRE}(t_i)$ as the set of direct precedents of t_i, i.e., $\text{PRE}(t_i) = \{t \in T \mid t \prec t_i\}$, the earliest starting time $\text{EST}(t_i)$ can be computed by calculating the maximum of the earliest finishing times of the predecessors, except if the release time is later, in which case the release time is used as earliest starting time. Hence, we have:

$$\text{EST}(t_i) = \max(\{\text{EST}(t) + l(t) \mid t \in \text{PRE}(t_i)\} \cup \{a(t_i)\}) \tag{1}$$

The latest starting time $\text{LST}(t_i)$ can be defined in similar fashion, using the set of successors $\text{SUC}(t_i) = \{t \in T \mid t_i \prec t\}$ of a task t_i:

$$\text{LST}(t_i) = \min(\{\text{LST}(t) - l(t_i) \mid t \in \text{SUC}(t_i)\} \cup \{d(t_i) - l(t_i)\} \tag{2}$$

To compute these starting times efficiently, we use a topological ordering of the underlying graph $(T, \prec)$. This ensures that each time point is only updated once. Since the topological ordering can be computed in linear time [5], earliest and latest starting times for all tasks can be computed in $O(n)$ time.

Example 3 *One topological ordering for the example is* $[t_4, t_1, t_2, t_3, t_5]$*. Using this ordering, the earliest start times are computed as follows:*

- $\text{EST}(t_4) = \max(\{\text{EST}(t) + l(t) \mid t \in \emptyset\} \cup \{a(t_4)\}) = a(t_4) = 1$
- $\text{EST}(t_1) = \max(\{\text{EST}(t) + l(t) \mid t \in \emptyset\} \cup \{a(t_1)\}) = a(t_1) = 0$
- $\text{EST}(t_2) = \max(\{\text{EST}(t) + l(t) \mid t \in \{t_1\}\} \cup \{a(t_2)\}) = \text{EST}(t_1) + l(t_1) = 1 + 1 = 2$
- $\text{EST}(t_3) = \max(\{\text{EST}(t) + l(t) \mid t \in \{t_2\}\} \cup \{a(t_3)\}) = \text{EST}(t_2) + l(t_2) = 2 + 1 = 3$
- $\text{EST}(t_5) = \max(\{\text{EST}(t) + l(t) \mid t \in \{t_3, t_4\}\} \cup \{a(t_5)\}) = \text{EST}(t_3) + l(t_3) = 3 + 1 = 4$

In similar fashion, the latest start times can be computed: $\text{LST}(t_1) = 7$*,* $\text{LST}(t_2) = 8$*,* $\text{LST}(t_3) = 10$*,* $\text{LST}(t_4) = 9$ *and* $\text{LST}(t_5) = 11$*.*

```
1  q ← ∅
2  while ∃t ∈ T do
3  |  select starting time σ(t) ∈ [EST(t), LST(t)]
4  |  for t_i ∈ SUC(t) do
5  |  |  a(t_i) ← max(a(t_i), σ(t) + l(t))
6  |  |  if a(t_i) was updated then q ← q ∪ t_i
7  |  end
8  |  UPDATE-EST(q)
9  |  q ← ∅
10 |  for t_i ∈ PRE(t) do
11 |  |  d(t_i) ← min(d(t_i, σ(t))
12 |  |  if d(t_i) was updated then q ← q ∪ t_i
13 |  end
14 |  UPDATE-LST(q)
15 |  q ← ∅
16 |  remove t from T
17 end
```

Algorithm 1: FIND-SOLUTION

```
Input: q: queue of nodes to be updated
1  while q ≠ ∅ do
2  |  let t_i be the topologically lowest task in q
3  |  EST(t_i) = max({EST(t) + l(t) | t ∈ PRE(t_i)} ∪ {a(t_i)})
4  |  if EST(t_i) was updated then q ← q ∪ SUC(t_i)
5  |  remove t_i from q
6  end
```

Algorithm 2: UPDATE-EST

```
Input: q: queue of nodes to be updated
1  while q ≠ ∅ do
2  |  let t_i be the topologically highest task in q
3  |  LST(t_i) = min({LST(t) − l(t_i) | t ∈ SUC(t_i)} ∪ {d(t_i) − l(t_i)}
4  |  if LST(t_i) was updated then q ← q ∪ PRE(t_i)
5  |  remove t_i from q
6  end
```

Algorithm 3: UPDATE-LST

2.3 Finding arbitrary solutions

Like in STNs, given these minimal and maximal starting times for the tasks, we can construct arbitrary solutions using FIND-SOLUTION (see Algorithm 1). Such an arbitrary solution algorithm can be used during execution: actual start times might be altered due to external factors, the algorithm can update the schedule when the actual start time of tasks becomes known. This algorithm begins by fixing the start time $\sigma(t)$ of an arbitrary task t, within the interval $[\text{EST}(t), \text{LST}(t)]$. Fixing this start time has consequences for the predecessors and successors of t: any predecessors must be finished at $\sigma(t)$, and any successors can only start after $\sigma(t)$. Therefore, the algorithm adjusts the due time of the predecessors and the release time of the successors, such that the due time is at or before $\sigma(t)$, and the release time is at or past $\sigma(t)$. Then, the procedures UPDATE-EST and UPDATE-LST (see Algorithm 2 and 3, respectively) are used to propagate the effects of the changed due/release times through the network. The task for which the start time was decided is removed from the network, and the algorithm starts over again until all start times have been decided.

The propagation through the network after choosing each $\sigma(t) \in [\text{EST}(t), \text{LST}(t)]$ ensures that future task start times are compatible with $\sigma(t)$. In the worst case, $O(n^2)$ time is needed to find a solution. If the tasks to which start times are assigned are chosen in a sequence respecting their topological ordering, the

propagation can be combined with the start time assignment. This reduces the time complexity of finding a solution to $O(n)$.

3 Delay management

As noted in the introduction, being able to cope with delays during execution of the schedule is important. In this section, we will first introduce the general concept of a delay and how to cope with them during execution. Then, the reverse is done, to calculate how sensitive to delay the schedule is. Finally, probabilistic techniques are introduced to analyze the schedule using advance information on delay distribution.

A delay is modeled as additional time $\delta(t_i)$ needed for the execution of the task, increasing the task length $l(t_i)$ to $l(t_i) + \delta(t_i)$. If the delay is sufficiently large, parts of the network might have to be updated: the earliest starting times of some or all of t_i's successors might need to be increased, and the latest starting time of t_i and some or all of its predecessors might need to be decreased. To update the earliest starting times, UPDATE-EST(SUC(t_i)) can be used. Similarly, to update the latest starting times, UPDATE-LST(t_i) can be used.

Example 4 *In our example described earlier, suppose t_2 suffers a delay of one hour, $\delta(t_2) = 1$, such that $l(t_2)$ increases to three.* UPDATE-EST *is called with $q = \text{SUC}(t_2) = \{t_3\}$. This procedure starts by updating $\text{EST}(t_3)$ to $\text{EST}(t_2) + l(t_2) = 1 + 3 = 4$. Then, $\text{SUC}(t_3) = \{t_5\}$ is added, and immediately processed: $\text{EST}(t_5)$ is set to $\text{EST}(t_3) + l(t_3) = 4 + 1 = 5$. Since $\text{SUC}(t_5) = \emptyset$ and q is empty as well, processing stops.*

With these procedures, a delay in a task can be propagated in $O(n)$ time, since every node is at most visited once. While a full recomputation of earliest and latest start times can also be performed in $O(n)$, the procedure described here is, in general, faster, since not all nodes have to be visited. The exact speedup however depends on the structure of the network. In the example above, the speedup is very small: except t_1, all nodes have to be visited.

In some cases a delay is sufficiently large to violate one or more of the constraints in the network. This can fortunately be easily detected by making use of the computed latest start time of the delayed task: if the new earliest start time is later than the latest start time, one or more due times will be violated. Since the latest start times are pre-computed, such a violation can be detected in constant time.

3.1 Delay resistance

The idea outlined in the previous paragraph can also be applied in reverse: instead of waiting on a delay and checking if it can be absorbed by the schedule, we can calculate in advance the level of *delay resistance* $\delta^{\perp}(t_i) = \text{LST}(t_i) - \text{EST}(t_i)$, i.e., the time which can maximally be absorbed by the schedule before due time violations will occur. If this is done for each task, it is possible to gain insight in the robustness of the schedule, in terms of being able to handle various delays.

Example 5 *For the example problem, we can compute that $\delta^{\perp}(t_1) = \delta^{\perp}(t_5) = 3$, and $\delta^{\perp}(t_2) = \delta^{\perp}(t_3) = \delta^{\perp}(t_4) = 1$. From this, it is clear that the schedule is less sensitive to delays in t_1 and t_5 than to delays in t_2, t_3 and t_4. During execution, this can be taken into account by extra monitoring of these last three tasks, or even allocation of extra resources.*

Next to causing due time violations, delays can extend the minimal makespan MM of a schedule. If we define the critical starting time $\text{CST}(t_i)$ as

$$\text{CST}(t_i) = \min(\{\text{CST}(t) - l(t_i) \mid t \in \text{SUC}(t_i)\} \cup \{MM - l(t_i)\}) \tag{3}$$

and propagate this through the network, we know for each task the latest time at which it can be started without extending the minimal makespan. The delay level $\delta^{+}(t_i) = \text{CST}(t_i) - \text{EST}(t_i)$ then gives an indication of the delay resistance each task has in the schedule with respect to the minimal makespan. Also, instead of the critical path[1], we can define a set of critical tasks $T_c = \{t_i \in T \mid \delta^{+}(t_i) = 0\}$, in which any delay will have immediate effect on the minimal makespan.

[1] The *critical path* in a schedule is a sequence of tasks with the longest total duration, hereby determining the minimal makespan of the schedule.

Table 1: Task length probability distributions

Task length	1	2	3	4	5
t_1	0.80	0.20			
t_2		0.60	0.30	0.10	
t_3	0.70			0.20	0.10
t_4		0.95	0.05		
t_5		0.85	0.10	0.05	

3.2 Using advance delay information

If advance information is known about the likelihood of certain delays occurring during schedule execution, it is worthwhile to be able to assess the delay-resistancy of a schedule, for example by determining the likelihood a given makespan will be met. To be able to do this, the advance delay information has to be represented in some form, and it has to be propagated through the network. One of the more intuitive ways of representing a potential delay is to include uncertainty in the task durations.

Discrete probability distributions seem like a good candidate to represent the task duration uncertainty, since they fit well with the generally sparse character of the available information. While it would be possible to fit continuous distributions to the task duration information, this tends to overrepresent the available information. Additionally, discrete distributions make it possible to simplify the modeling of inspection-and-repair tasks, which are very common in maintenance. These tasks represent the inspection of a component, with the chance that a repair needs to be performed. While modeling the repair tasks as an optional tasks with some probability is possible, discrete distributions enable us to model both task in one single task, with a bi-modal probability distribution.

An additional argument for using discrete probability distributions is that whatever representation of this uncertainty is chosen, it must be able to support the operations needed to propagate it through the network. Calculating the maximum end time of the predecessors for each task is very difficult using continuous probability distributions; using discrete distributions this problem is avoided.

3.3 Operations on continous probability distributions

To support the use of continous probability distributions, we must define the basic operations for task schedule calculations. If we assume two random variables X_1 and X_2, with distribution functions f and g, respectively, defined on some finite subset of $\mathbb{N}$, the needed operations can be defined as follows:

- For the maximum of two independent random variables, the cumulative distribution function is used, since the maximum of two cumulative distributions can easily be calculated: $Pr(\max(X_1, X_2) \leq x) = Pr(X_1 \leq x)Pr(X_2 \leq x)$.
- Similarly, the minimum can be calculated: $Pr(\min(X_1, X_2) \leq x) = 1 - Pr(\min(X_1, X_2) > x) = 1 - Pr(X_1 > x)Pr(X_2 > x)$.
- For the sum, the convolution $(f * g)$ of the two distribution functions f and g is calculated:

$$(f * g)(n) = \sum_{m \in \text{supp}(f), m-n \in \text{supp}(g)} f(m)g(n-m), \tag{4}$$

  where $\text{supp}(f)$ denotes the support of f, i.e., $\text{supp}(f) = \{x \mid f(x) \neq 0\}$.

Example 6 *To demonstrate the operations, assume X_1 has distribution $f(1) = 0.9$, $f(2) = 0.1$ and $f(n) = 0$ otherwise, and X_2 has distribution $g(1) = 0.8$, $g(2) = 0.15$, $g(3) = 0.05$ and $g(n) = 0$ otherwise. $X_1 + X_2$ now has distribution $f * g$, with $(f*g)(2) = f(1)g(1) = 0.72$, $(f*g)(3) = f(2)g(1) + f(1)g(2) = 0.215$, etc.*

To calculate $\max(f, g)$, we first need the cumulative distribution functions F and G of X_1 and X_2: $F(0) = 0$, $F(1) = 0.9$ and $F(n) = 1$ for $n \geq 2$; and $G(0) = 0$, $G(1) = 0.8$, $G(2) = 0.95$ and $G(n) = 1$ for $n \geq 3$. Then, we can compute $\max(F, G)$: $\max(F, G)(1) = F(1)G(1) = 0.72$, $\max(F, G)(2) = F(2)G(2) = 0.95$, etc. Converting this back to a probability distribution function, we obtain $\max(f, g)(1) = 0.72$, $\max(f, g)(2) = 0.23$ and $\max(f, g)(3) = 0.05$.

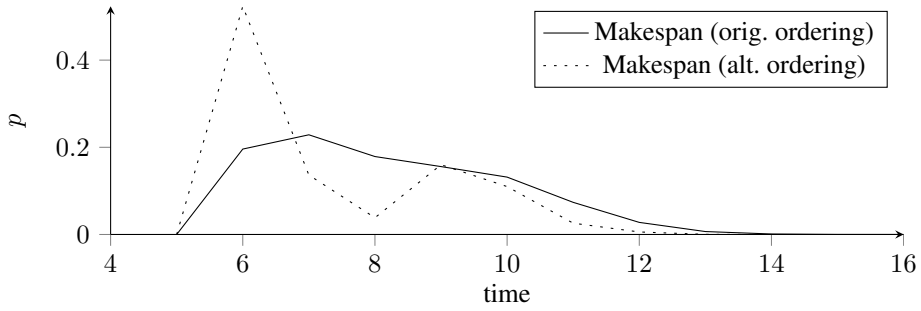

Figure 1: Probability distribution of the makespan of the schedule

Note that the maximum operation assumes independently distributed random variables; in actual schedules this condition might not always hold. Still, it is believed that for most schedules, the error introduced by this assumption is marginal.

To apply this to the example problem introduced earlier, consider the task length probability distributions in Table 1. Most tasks have the peak of their distribution around the original task length. Task t_3 is a slight exception, it is modeled as an inspection task. In most ($p = 0.70$) of the cases, this task will last only one hour. But in some cases, the inspection performed will lead to a significant amount of additional work, hence the peak around $l(t_3) = 4$. Using the data from Table 1, it can be calculated that the expected value of the makespan is 8.17. The probability that the due time of the first train is violated is none; for the second train this probability is $p = 0.0001$. The probability distribution for the makespan is shown in Figure 1.

While the precedence ordering decision $t_3 \prec t_5$ made sense using the original data, with the more detailed task duration data from Table 1, it might make more sense to reverse this ordering. Using $t_5 \prec t_3$, the expected makespan slightly decreases to 7.30. While the due time for the second train will now always be met, the probability for a due time violation for the first train is now $p = 0.006$. The new probability distribution is also shown in Figure 1.

3.4 Computational complexity

Using discrete probability distributions obviously raises the computational complexity. If it is assumed that all discrete probability functions are defined on equidistant points, the size of their support can be used to represent the complexity of calculations on the distributions. For each sum $X_1 + X_2$, the support size of $(f * g)$ is defined as

$$\begin{aligned} \big|\operatorname{supp}(f * g)\big| &= \big|[\min(\operatorname{supp}(f)) + \min(\operatorname{supp}(g)), \max(\operatorname{supp}(f)) + \max(\operatorname{supp}(g))]\big| \\ &= \big|\operatorname{supp}(f)\big| + \big|\operatorname{supp}(g)\big|. \end{aligned}$$

For each point in the support of $(f * g)$, Equation 4 has to be computed. The number of operations is therefore bounded by $\big(\big|\operatorname{supp}(f)\big| + \big|\operatorname{supp}(g)\big|\big)^2$.

For $\max(X_1, X_2)$, we have

$$\begin{aligned} \big|\operatorname{supp}(\max(f, g))\big| &= \big|[\max(\min(\operatorname{supp}(f)), \min(\operatorname{supp}(g))), \max(\operatorname{supp}(f) \cup \operatorname{supp}(g))]\big| \\ &< \max(|\operatorname{supp}(f)|, |\operatorname{supp}(g)|). \end{aligned}$$

Each point in the support of $\max(f, g)$ only needs one multiplication, hence the number of operations is bounded by $\max(|\operatorname{supp}(f)|, |\operatorname{supp}(g)|)$.

If the network has the form of a linear sequence of tasks with $p = |\prec|$ precedence constraints, the worst-case support size for the makespan is $s_{\max} = sp$, if all task lengths have equal support size s. To arrive at this makespan, p additions of probability distributions have to be performed, of cost $4s_{\max}^2$. Over all additions, the total cost is thus $O(s_{\max}^2 p) = O(s^2 p^3)$. Calculating the maximum does not increase the support size of the probability distributions. In the worst case, every precedence link induces the computation of a maximum; the total cost of determining all maxima is therefore $O(s_{\max} p) = O(sp^2)$.

The main cause of the higher computational complexity is the dependence of the support size of the distribution on the number of precedence constraints in the graph. A very simple remedy is to limit the size of the support beforehand, trading a bit of accuracy for a gain in computational speed. If $s_{\max}$ is limited to

some constant c, the addition complexity reduces to $O(s_{\max}^2 p) = O(p)$. If a calculation results in probability mass outside the maximum support interval, this probability mass is ignored; the probability mass inside the support interval must however be normalized to sum to one.

3.5 Determining consistency

When using normal numbers, determining if the schedule is consistent (i.e., if all due time constraints are respected) is rather straightforward: if it holds for all tasks t that $\text{EST}(t) \leq \text{LST}(t)$, it is easy to see that start times for the tasks can be chosen in such a way that all tasks are started before their latest starting times, and as such will meet their due times. Using discrete probability distribution, the comparison somewhat more involved. Instead of calculating a yes or no answer, we can compute the probability $\Pr(X_1 \leq X_2) = \Pr(X_1 - X_2 \leq 0)$ that the due time will be met, where X_1 is the random variable representing the distribution of the earliest start time of t, and X_2 of the latest start time.

4 Discussion

Summarizing, we have introduced a simple formalism with enough expressive power to support the scheduling of NedTrain workshop activities. Due to its simplicity, manipulation of schedules can be performed very efficiently. In particular, it has been shown that the schedule can be updated quickly when delays occur during execution. To make use of advance delay information, the basic operations needed for schedule manipulations are redefined in terms of discrete probability distribution.

Next to updating the schedule to incorporate delays, the algorithms described in this paper can be used for fully dynamic schedule manipulation. If a constraint $t_a \prec t_b$ has been added or removed, the earliest and latest start time information can be updated using the same process as used for a delayed task. For the earliest starting times, UPDATE-EST(t_b) can be used; for the latest starting times, UPDATE-LST(t_a) can be used.

If a task t_c needs to be inserted between two existing, dependent, activities t_a and t_b in the STN, the delay calculations described above can be used as well. The insertion is only possible if $l(t_c) \leq \delta^{\perp}(t_a)$, similarly, the makespan is not affected if $l(t_c) \leq \delta^{+}(t_a)$. Updating the constraints of dependent tasks can be done by propagating a "delay" of $l(t_c)$, using the procedure described earlier.

Note also that, if multiple options for placing a task are available, the delay levels of the various tasks can be used to effectively guide some higher-level algorithm in choosing a suitable location; a location where the slack is large enough to absorb the new task completely is usually to be preferred over a location where (almost) all of the task length has impact on the makespan of the schedule.

An important future research direction is the combination of the task schedule framework with constraint posting algorithms ([1, 2]). The efficient operations, combined with the possibility to add *and* retract constraints, make this formalism very suitable to better explore the search space of resource-constrained schedules. Additionally, the delay level information and the probabilistic task length representation represent useful sources of information for new and improved heuristics. Lastly, it is important to investigate the error caused by the assumption of independently distributed task lengths in calculating the maximum.

References

[1] Amedeo Cesta, Angelo Oddi, and Stephen F Smith. Profile-based algorithms to solve multiple capacitated metric scheduling problems. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 214–223, 1998.

[2] Amedeo Cesta, Angelo Oddi, and Stephen F Smith. Iterative flattening: A scalable method for solving multi-capacity scheduling problems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 742–747, 2000.

[3] Rina Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[4] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

[5] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5:558–562, November 1962.