On the parametric complexity of schedules to minimize tardy tasks

Michael R. Fellows, Catherine McCartin
School of Mathematical and Computing Sciences, Victoria University, Wellington, New Zealand

Abstract

Given a set $T$ of tasks, each of unit length and having an individual deadline $d(t) \in Z^+$, a set of precedence constraints on $T$, and a positive integer $k \leq |T|$, we can ask “Is there a one-processor schedule for $T$ that obeys the precedence constraints and contains no more than $k$ late tasks?” This is a well-known $NP$-complete problem.

We might also inquire “Is there a one-processor schedule for $T$ that obeys the precedence constraints and contains at least $k$ tasks that are on time i.e. no more than $|T| - k$ late tasks?”

Within the framework of classical complexity theory, these two questions are different instances of the same problem. However, within the recently developed framework of parameterized complexity theory, they give rise to two separate problems which may be studied independently of one another.

We investigate these problems from the parameterized point of view. We show that, in the general case, both these problems are hard for the parameterized complexity class $W[1]$.

In contrast, in the case where the set of precedence constraints can be modelled by a partial order of bounded width, we show that both these problems are fixed parameter tractable.

1 Introduction

Scheduling to minimize tardy tasks is a well-known problem that has several related variations. The majority of these are known to be $NP$-complete.

In this article we concentrate on the following scenario. We are given a set $T$ of tasks, each of unit length and having an individual deadline $d(t) \in Z^+$, a set of precedence constraints on $T$, and a positive integer $k \leq |T|$.

We ask “Is there a one-processor schedule for $T$ that obeys the precedence constraints and contains no more than $k$ late tasks?”

This problem is $NP$-complete even if the precedence constraints are modelled by a partial order consisting only of chains, i.e. each task has at most one immediate predecessor and at most one immediate successor. It can be solved in polynomial time if $k = 0$, or if the set of precedence constraints is empty.
We also make the dual inquiry “Is there a one-processor schedule for $T$ that obeys the precedence constraints and contains at least $k$ tasks that are on time i.e. no more than $|T| - k$ late tasks?”

Within the framework of classical complexity theory, these two questions are different instances of the same problem. However, within the recently developed framework of parameterized complexity theory, they give rise to two separate problems which may be studied independently of one another.

We model the jobs and their precedence constraints as a finite ordered set, or partial order. A schedule is a linear extension of this partial order.

A linear extension $L = (≤, P)$ of a partial order $(P, ≤)$ is a total ordering of the elements of $P$ in which $a < b$ in $L$ whenever $a < b$ in $P$.

If $(P, ≤)$ is a partial order, a subset $X$ of $P$ is a chain iff any two distinct elements of $X$ are comparable, and an antichain iff no two distinct elements of $X$ are comparable.

By Dilworth’s theorem [5], the minimum number of chains which form a partition of $(P, ≤)$ is equal to the size of a maximum antichain. This number is the width of $(P, ≤)$, denoted $w(P)$.

In this article, we show that, in the general case, each of the problems defined by the questions above is hard for the fundamental parameterized complexity class $\text{W}[1]$. This means that, for either problem, there does not exist a constant $c$, such that for all fixed $k$, the problem can be solved in time $O(n^c)$, unless an unlikely collapse occurs in the $\text{W}$-hierarchy of parameterized complexity classes.

In contrast, in the case where the set of precedence constraints can be modelled by a partial order of bounded width, we show that both problems are fixed parameter tractable.

In the following section, we introduce the main concepts of parameterized complexity theory, and define the parameterized complexity class $\text{W}[1]$. In section 3, we show that each of the problems, in the general case, is $\text{W}[1]$-hard. In section 4, we show that the restricted version of each problem is fixed parameter tractable.

2 Parameterized Complexity

Many natural computational problems have input that consists of several elements of information. It is natural to consider some of these elements as a parameter and evaluate their relative contribution to the overall complexity of the problem. In many practical applications of computational problems only a small set of parameter values may be significant. Often, the size of the parameter chosen can be considered “very small” in comparison with the size of the main part of the input.

A framework in which to study this complexity issue has been developed in [1].

In classical complexity, a decision problem is specified by two items of information: the input to the problem, and the question to be answered. In parameterized complexity there are three parts to a problem specification: the
input to the problem, the aspects of the input that constitute the parameter, and the question.

**Definition 1 (Parameterized Language)** A parameterized language \( L \) is a subset \( L \subseteq \Sigma^* \times \Sigma^* \). If \( L \) is a parameterized language and \((x, y) \in L\) then we refer to \( x \) as the “main part” and \( y \) as the “parameter”.

In classical complexity, the notion of “good” behaviour is polynomial-time solvability. In parameterized complexity, the notion of good is fixed-parameter tractability.

**Definition 2 (Fixed-parameter Tractability)** A parameterized problem \( L \subseteq \Sigma^* \times \Sigma^* \) is fixed-parameter tractable (or FPT), if there is an algorithm that correctly decides, for input \((x, y) \in \Sigma^* \times \Sigma^*\), whether \((x, y) \in L\) in time \( O(f(k)n^c) \), where \( n = |x| \), \( k = |y| \), \( c \) is some constant independent of \( n \) and \( k \), and \( f \) is some arbitrary function based only on \( k \).

As with classical complexity, where this “good” behaviour can be achieved, the exponent in \( n \) is usually quite small.

However, there are many parameterized problems that apparently do not admit this “good” behaviour. We are faced, however, with the situation where proving parametric intractability for such problems would also settle the question \( P = NP \)? Since this is probably the most important and longstanding question of complexity theory, we settle instead for providing strong evidence of the likely intractability of such problems, under the assumption that \( P \neq NP \).

As with classical complexity, we can find evidence for parametric intractability by studying the appropriate notion of problem transformation.

**Definition 3 (Parametric Transformation)** A parametric transformation from a parameterized language \( L \) to a parameterized language \( L' \) is an algorithm that computes from input consisting of a pair \((x, k)\), a pair \((x', k')\) such that:

1. \((x, k) \in L\) if and only if \((x', k') \in L'\),
2. \(k' = g(k)\) is a function only of \( k\), and
3. the computation is accomplished in time \( f(k)n^c\), where \( n = |x| \), \( c \) is a constant independent of both \( n \) and \( k \), and \( f \) is an arbitrary function.

The essential property of parametric transformations is that if \( L \) transforms to \( L' \) and \( L', L' \in FPT \), then \( L \in FPT \). This leads naturally to a completeness program based on a hierarchy of parameterized complexity classes, called the W-hierarchy, of the form:

\[
FPT \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq W[SAT] \subseteq W[P] \subseteq AW[P] \subseteq XP
\]

The parameterized analogue of \( NP \) is \( W[1] \), and \( W[1] \)-hardness is the basic evidence that a parameterized problem is likely not to be fixed-parameter tractable. The \( k \)-STEP HALTING PROBLEM FOR NON-DETERMINISTIC TURING MACHINES is \( W[1] \)-complete [4]. Since the \( q(n) \)-STEP HALTING PROBLEM is essentially the defining problem for \( NP \), the analogy is very strong.
3 Parametric complexity of schedules to minimize tardy tasks

**k-LATE TASKS**

- **Instance**: A set $T$ of tasks, each of unit length and having an individual deadline $d(t) \in \mathbb{Z}^+$; and a set $(P, \preceq)$ of precedence constraints on $T$.
- **Parameter**: A positive integer $k \leq |T|$.
- **Question**: Is there a one-processor schedule for $T$ that obeys the precedence constraints and contains no more than $k$ late tasks?

**k-TASKS ON TIME**

- **Instance**: A set $T$ of tasks, each of unit length and having an individual deadline $d(t) \in \mathbb{Z}^+$; and a set $(P, \preceq)$ of precedence constraints on $T$.
- **Parameter**: A positive integer $k \leq |T|$.
- **Question**: Is there a one-processor schedule for $T$ that obeys the precedence constraints and contains at least $k$ tasks that are on time?

As mentioned in the introduction, within the framework of classical complexity theory, the two questions posed are different instances of the same $NP$-complete problem. In the first case we ask for a schedule with no more than $k$ late tasks, in the second case we ask for a schedule with no more than $|T| - k$ late tasks.

From the parameterized complexity point of view they give rise to two separate problems, each with parameter $k$. It is true that $T$ has a schedule with at least $k$ tasks on time if $T$ has a schedule with at most $|T| - k$ late tasks. This gives us a polynomial-time reduction, transforming an instance $(T, k)$ of $k$-TASKS ON TIME into an instance $(T, k')$ of $k$-LATE TASKS. However, this is not a parametric transformation, since $k' = |T| - k$ is not purely a function of $k$.

Indeed, while we show that both these problems are $W[1]$-hard, it is possible that they inhabit quite separate regions of the $W$-hierarchy.

Our results rely on the following theorem from [3].

**Theorem 1** $k$-CLIQUE is complete for the class $W[1]$.  

$k$-CLIQUE is defined as follows:

- **Instance**: A graph $G = (V, E)$.
- **Parameter**: A positive integer $k$.
- **Question**: Is there a set of $k$ vertices $V' \subseteq V$ that forms a complete subgraph of $G$ (that is, a clique of size $k$)?
**Theorem 2** \(k\text{-LATE TASKS}\) is \(W[1]\text{-hard}\).

**Proof:**

We transform from \(k\text{-CLIQUE}.

\[(G = (V, E), k) \rightarrow (T, (P, \leq), k')\text{ where } k' = k(k-1)/2 + k.\]

We set up \(T\) and \((P, \leq)\) as follows:

For each vertex \(v\) in \(V\), \(T\) contains a task \(t_v\). For each edge \(e\) in \(E\), \(T\) contains a task \(s_e\). The partial order relation constrains any edge task to be performed before the two vertex tasks corresponding to its endpoints. So we have a two-layered partial order as shown below.

We set the deadline for edge tasks to be \(|E| - k(k-1)/2\), the deadline for the vertex tasks to be \(|E| - k(k-1)/2 + (|V| - k)\). At most \(k(k-1)/2 + k\) tasks can be late. At least \(k(k-1)/2\) edge tasks will be late. The bound is only achieved if \(k(k-1)/2\) tasks in the top row are late and they only block \(k\) tasks in the bottom row.

Thus, a YES for an instance \((T, (P, \leq), k')\) means that the \(k(k-1)/2\) edge tasks that are late only block \(k\) vertex tasks, and these correspond to a clique in \(G\). \(\square\)

![Figure 1: Gadget for \(k\text{-LATE TASKS}\) transformation.](image)

**Theorem 3** \(k\text{-TASKS ON TIME}\) is \(W[1]\text{-hard}\).

**Proof:**

Again, we transform from \(k\text{-CLIQUE}.

\[(G = (V, E), k) \rightarrow (T, (P, \leq), k')\text{ where } k' = k(k+1)/2.\]

We set up \(T\) and \((P, \leq)\) as follows:

For each vertex \(v\) in \(V\), \(T\) contains a task \(t_v\). For each edge \(e\) in \(E\), \(T\) contains a task \(s_e\). The partial order relation constrains any edge task to be performed after the two vertex tasks corresponding to its endpoints. So we have a two-layered partial order as shown below.

We set the deadline for vertex tasks to be \(k\), and set the deadline for edge tasks to be \(k(k+1)/2\). Therefore, we can only do at most \(k\) vertex tasks on time, then \(k(k-1)/2\) edge tasks on time.
Thus, a YES for an instance \((T,(P,\leq),k')\) means that the \(k(k-1)/2\) edge tasks done on time fall below the \(k\) vertex tasks done on time, and these correspond to a clique in \(G\).

\[
\begin{array}{c}
\text{deadline } k \\
\text{deadline } \frac{k(k+1)}{2}
\end{array}
\begin{array}{c}
t_1 \\
_2 \\
t_3 \\
t_i \\
t_m
\end{array}
\begin{array}{c}
s_1 \\
s_2 \\
s_i \\
s_i \\
s_q
\end{array}
\]

Figure 2: Gadget for \(k\)-TASKS ON TIME transformation.

4 Parametric complexity of bounded-width schedules to minimize tardy tasks

It is important to note that a given classical problem can be examined under many different parameterizations, some of these may lead to fixed-parameter tractability, and some may lead to various levels of intractability. The two results above rely heavily on the \(O(n)\) width of the constructed partial order. This leads to consideration of partial orders in which the width is bounded, and treated as a parameter.

We now recast the problems using a new parameter \((k,m)\), \(k \leq \lceil 7 \rceil\), \(m = w(P)\). Under this parameterization, we find that both problems are fixed-parameter tractable (FPT).

FPT algorithm design has a distinctive toolkit of positive techniques, including two important elementary methods, "bounded search tree" and "reduction to problem kernel". The two algorithms given here illustrate these techniques.

4.1 FPT algorithm for \(k\)-LATE TASKS

We first compute \(w(P)\) and decompose \((P,\leq)\) into a set of \(w(P)\)-many chains. We can either fix our choice of \(m\), so that \(m = w(P)\), or abort if \(w(P)\) is outside the range we wish to consider.

To compute \(w(P)\) and a decomposition of \((P,\leq)\) requires the following:

1. \(T\) is the ground set of \(P\). Form a bipartite graph \(K(P)\) whose vertex set consists of two copies \(T'\) and \(T''\) of \(T\). In the graph, let \((x',y')\) be an edge iff \(x < y\) in \(P\). Then a maximum matching in \(K(P)\) corresponds uniquely to a minimum partition of \((P,\leq)\) into chains [6].

Any bipartite matching algorithm can be used to find the optimum partition, the best known algorithm being \(O(n^{2.5})\) [7].

Now let \(n = \lceil 7 \rceil\). There are \(n\) tasks to schedule, and one of these has to go in the \(nth\) position in the final ordering. This must be some maximal element in the partial order \((P,\leq)\), since it cannot be an element that has a successor.
There are at most \( m \) maximal elements to choose from, so consider these. If there is a maximal element with deadline \( \geq n \) then put it in the \( n \)th position. This will not affect the rest of the schedule adversely, since no other element is forced into a later position by this action. We now ask “Is there a schedule for the remaining \( n-1 \) elements that has at most k late tasks?”.

If there is no maximal element with deadline \( \geq n \) then one of the maximal elements in \((P, \leq)\) has to go in the \( n \)th position and will be late, but which one should be chosen? Delaying on particular chains may be necessary in obtaining an optimal schedule. For example, in the \( W[1] \)-hardness result given in the last section, it is important to leave the clique elements until last.

In this case, we begin to build a bounded search tree. Label the root of the tree with \( \emptyset \). For each of the (at most \( m \)) maximal elements produce a child node, labelled with a partial schedule having that element in the \( n \)th position, and ask “Is there a schedule for the remaining \( n-1 \) tasks that has at most \( k-1 \) late tasks?”.

The resulting tree will have at most \( m^k \) leaves. We branch only when forced to schedule a late task, which can happen at most \( k \) times, and each time produce at most \( m \) children.

On each path from root to leaf at most \( O(mn) \) computations are done.

We can use an adjacency list representation of \((P, \leq)\), with a separate inlist and outlist for each element, containing respectively the elements covered by, and covering, that element. Thus, a maximal element is one whose outlist is empty.

Each element is inserted into the schedule after considering at most \( m \) maximal elements of the current partial order.

After scheduling a maximal element and removing it from \((P, \leq)\), we can find any new maximal elements by looking at the inlist of the removed element and checking for any element whose outlist is \( m \), so this routine takes time \( O(m) \).

Thus, the running time of the algorithm is \( O(m^{k+1}n + n^{2.5}) \).

4.2 FPT algorithm for \( k \)-TASKS ON TIME

As before, we first compute \( w(P) \) and decompose \((P, \leq)\) into a set of \( w(P) \)-many chains.

This time, we will reduce the input to a problem kernel. That is, a new problem instance whose size is bounded by a function of the parameter.

We first run through each chain separately. For each element in a chain, we count the number of its predecessors in all the chains and check whether this is less than its deadline.

If we find any chain with \( k \) elements that could be scheduled on time we are done. We schedule this chain, along with all the necessary predecessors from other chains, as an initial segment of the schedule.

Otherwise, there are \( < mk \) elements that could possibly be on time in the final schedule. There less than \( m^{mk} \) ways to order these “possibly on-time”
elements relative to one another (some of these orderings may be incompatible with the original constraints). Their order relative to others in the same chain is fixed, so we have at most $m$ choices for the one which occurs first in the final schedule, at most $m$ choices for the one which occurs second, and so on.

We can try out all the possible orderings, throwing in the necessary predecessors along the way. If one of the orderings is successful, we report "yes", otherwise "no".

To check each ordering requires $O(n)$ computations.

We again use the adjacency list representation of $(P, \leq)$, described above.

We work backwards from each of the elements of the ordering in turn, scheduling the predecessors of the first element, followed by the first element itself, then predecessors of the second that do not lie below the first, and so on.

This can be done by recursively checking the inlists of each element and its predecessors. No element of $(P, \leq)$ will be considered more than once, since, if we encounter an already-scheduled element, we can assume that all its predecessors have been dealt with. If we find an element of the ordering has already been scheduled when we come to consider it, we can discard this particular ordering immediately, as it must be incompatible with the original constraints.

The same approach can be used in the initial counting process, and in scheduling a chain as an initial segment, if a suitable candidate is discovered.

Thus, the running time of the algorithm is $O(m^{m^k}n + n^{25})$.

References


