Universität Bremen

Fachbereich 3: Mathematik und Informatik

Master's Thesis

Conditional Directed Acyclic Graphs: On the Complexity of Computing the Worst-Case Execution Time

Jens Schlöter

Matriculation No. -

June 20, 2019

First Examiner: Prof. Dr. Nicole Megow
Second Examiner: Prof. Dr. Carsten Lutz
Advisor: Prof. Dr. Nicole Megow

Declaration of Authorship

I hereby confirm that the thesis I am submitting is my own original work. Any use other materials and works of other authors is properly acknowledged at their point of use.

Bremen, June 20, 2019

_____

Jens Schlöter

Contents

Chapter 1

Introduction

Scheduling is the allocation of limited resources to tasks over time with the goal to optimize some objective. Tasks could for example be threads in a computer program and the resources a number of processors that are available to execute the threads. The goal then could be to assign the threads to processors with the objective to minimize the time needed to execute all threads.

In practice, scheduling problems are of significant importance. According to Pinedo [50], scheduling is an important factor in manufacturing, production and information processing systems as well as in transportation, distribution and other service industries. See [18] and [50] for more examples of practical scheduling applications.

On the theoretical side, scheduling problems have been extensively studied. An example for a classical scheduling problem, that can be used to model a number of practical applications, is $P|prec|C_{\max}$. In its decision problem variant, a set of jobs with processing times in form of a directed acyclic precedence constraint graph, a number of parallel identical machines and a deadline are given. The goal then is to decide whether all jobs can be executed within the deadline using the machines while respecting the precedence constraints. The semantics of the precedence constraint graph is that if there is an edge from job j to job j', then job j' can only be started after job j has been completed. The problem was shown to be strongly NP-hard by Ullman [58], but can be approximated using *list scheduling* [30, 37]. The algorithm uses some arbitrary *fixed-priority order* over all jobs and, whenever a machine is free, schedules the available job with the highest priority. This approach is also called *fixed-priority scheduling*.

While $P|prec|C_{\max}$ can be used to model a number of practical applications, there are relevant scenarios that cannot be expressed. As the problem assumes that all of the given jobs need to be executed, it needs to be exactly known which jobs are going to be processed when modeling application scenarios. In contrast, practical tasks that need to be executed often are computer programs whose source code contains control flow instructions, like for example `if-then-else` statements. Thus, when modeling such computer programs, it might not be known which parts of the source code will actually be executed and it therefore is unclear how they should be modeled using a directed

acyclic precedence constraint graph.

To overcome this problem, the *conditional DAG* model was proposed in [9] respectively [46]. This model again represents jobs using a directed acyclic precedence constraint graph but additionally allows the usage of so-called *conditional nodes*, where the semantics of a conditional node is that exactly one of its successors needs to be processed. Figure 1.1 exemplary shows the modeling of source code with a conditional DAG, where each conditional node is depicted by a square.

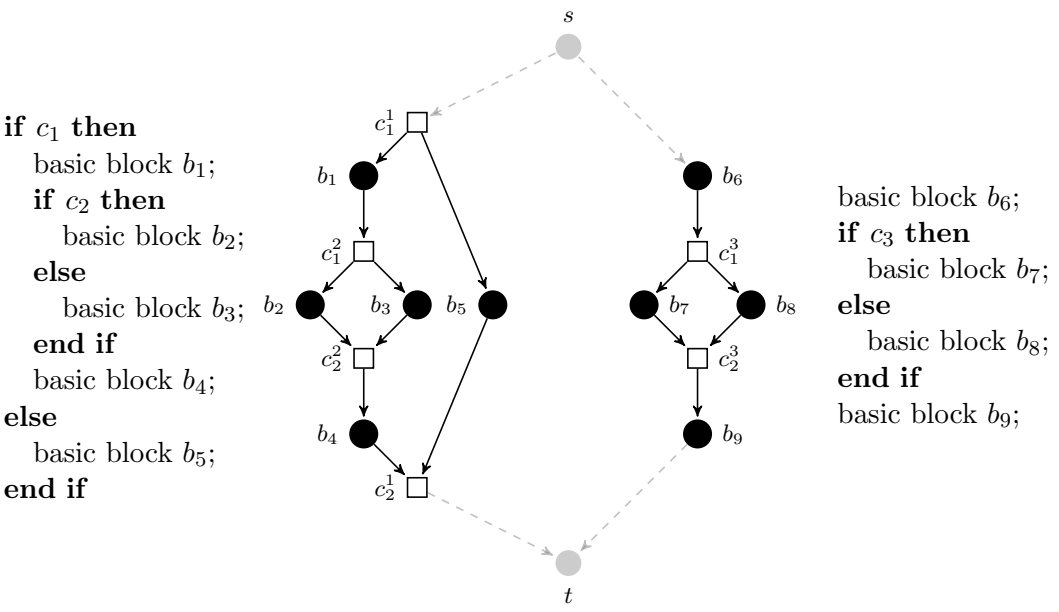


Figure 1.1 Two example source code excerpts and their representation as conditional DAGs. Each basic block b_i represents a sequence of statements that is guaranteed to be executed sequentially and each c_i represents some boolean expression. The dummy source s and sink t show how the parallel execution of two source code excerpts can be represented with a single connected conditional DAG.

When using the conditional DAG model to represent jobs, there are several possible subsets of jobs that could eventually be executed. In the right component of the example in Figure 1.1 for example, either the jobs $\{b_6, b_7, b_9\}$ or $\{b_6, b_8, b_9\}$ are going to be executed. Each of those subsets defines a subgraph that could eventually be processed. We call such subgraphs *realizations*. In this thesis, the variant of $P|prec|C_{\max}$ that uses a conditional DAG to represent the jobs is considered. In this variant, the goal is to determine whether each realization that could potentially be executed can be processed within the deadline. As this variant is clearly a generalization of $P|prec|C_{\max}$, it is strongly NP-hard as well.

Furthermore, assigning a priority to each job and using fixed-priority scheduling is a fairly good approximation for non-conditional $P|prec|C_{\max}$. Therefore, the assumption that jobs are scheduled using fixed-priority scheduling or similar algorithms is common

in related research areas, as will be pointed out in Section 1.2. In contrast, even if we fix a priority order over all jobs in a conditional DAG and assume that the jobs will be scheduled using fixed-priority scheduling, the time needed to process the jobs still depends on which subset of jobs will actually be executed. This thesis considers the problem of computing the *worst-case execution time* of a conditional DAG under the assumption that the jobs are scheduled using fixed-priority scheduling with a given fixed-priority order. The worst-case execution time is the maximum amount of time needed to process any possible realization of a conditional DAG using the fixed-priority order. By computing the worst-case execution time of a conditional DAG, it is also possible to decide whether a conditional DAG can be executed within a given deadline under the same assumptions, which is also called a *schedulability test*.

The focus on fixed-priority scheduling can also be motivated by its usage in the context of real-time tasks. For example, [5, 34, 57] consider the computation of worst-case execution times for tasks that are scheduled using a fixed-priority order to analyze the schedulability of real-time tasks. In addition, [9, 11, 26, 46] consider the schedulability analysis of conditional real-time tasks in order to include conditional execution of source-code. As the problem, that is considered in this thesis, is a basic case of an fixed-priority schedulability analysis for conditional real-time tasks, complexity results and algorithms for the problem are relevant in that context.

1.1 Outline and Results

The following paragraphs outline the structure and content of this thesis and list the results that we were able to obtain. The rest of this chapter references related work and subsequently introduces preliminaries that are used during the course of the thesis. This includes graph and complexity theoretical basics as well as the conditional DAG model.

The second chapter considers the complexity of computing the worst-case execution time of a conditional DAG for a given fixed-priority order. In specific, the corresponding decision problem is shown to be strongly CoNP-hard in the general case. Additionally, an even more general variant of the problem that allows dependencies between different conditions is shown to be strongly CoNP-hard even if the jobs are processed by a single machine. This first two CoNP-hardness proofs (see Sections 2.2 and 2.3.1) are based on previous work and ideas by Alberto Marchetti-Spaccamela, Nicole Megow, Martin Skutella and Leen Stougie.

During the course of this first two proofs, we exploit a non-obvious relation between the computation of the worst-case execution time for a conditional DAG given a fixed priority order and the *list scheduling makespan maximization* problem (LS MAX). In LS MAX, an $P|prec|C_{\max}$ instance is given and the goal is to find the maximum makespan that can be achieved on the instance using a list scheduling algorithm. The identified relation

gives us a proof framework for variants of the worst-case execution time problem; by showing that LS MAX is NP-hard for special graph classes, we show that the worst-case execution time problem is CoNP-hard for the corresponding graph classes.

Using the proof framework, several special cases of the problem are considered and complexity results derived. For example, the problem is shown to still be CoNP-hard if only *series-parallel* conditional DAGs are considered. Additionally, we show that the problem remains CoNP-hard if each possible realization of the conditional DAG is a tree or a constant number of chain components. Furthermore, we show that the problem for general graph structures is still CoNP-hard even if *preemption* is allowed, i.e., even if the execution of jobs can be interrupted and continued at a later point in time. Finally, we will derive some hardness results for the case where the number of machines is constant.

In the third chapter, algorithms that compute the worst-case execution time of a conditional DAG for a given fixed-priority order are considered. First, the exact polynomial time algorithm for the special case where the conditional DAG is processed on a single machine is discussed. The algorithm was first introduced in [46] and relies on the independence of different (non-nested) conditions. Then, an exact pseudo-polynomial dynamic program for the special case of conditional DAGs whose realizations have a bounded width is derived.

Thereafter, approximations for the problem are discussed. First, a 2-approximation for the general problem is derived, that was first introduced in [46]. Subsequently, a *fully polynomial-time approximation scheme* (*FPTAS*) based on the previously introduced dynamic program is derived, for the case of conditional DAGs whose realizations have a bounded width, if a certain monotonicity property holds. The monotonicity condition formulates that an increased processing time for a single job cannot lead to a decreased makespan in the fixed-priority schedules. While the monotonicity property does not hold for general conditional DAGs, as shown by the example in [30, 37] which is also known as *Graham anomaly*, we show that it does hold for conditional DAGs if each realization is a constant number of chain components. It then follows, that the introduced family of algorithms is an FPTAS for those conditional DAGs as well.

Finally, another special case of the problem that places restrictions on the fixed-priority order is considered. We are able to observe that some CoNP-hard cases of the problem are solvable in polynomial time if the restrictions on the fixed-priority order hold.

The final chapter summarizes the results of this thesis, draws a conclusion and gives an outlook on possible future work.

1.2 Other Related Work

In many real-world applications time-critical jobs or tasks need to be executed on a shared uni- or multi- processor platform. As those tasks are time-critical, they need to

be completed within a limited amount of time and therefore the scheduling of the tasks is of special importance to ensure that all tasks finish within the available amount of time. While $P|prec|C_{\max}$ is one possible way of modeling such scenarios, the execution of time-critical tasks often occurs in real-time scenarios, where a finite number of recurrent tasks generates a (possibly infinitive) sequence of jobs that needs to be processed.

To model such scenarios, Liu and Layland [42] in 1973 proposed a model to represent the execution of different recurrent tasks on a single-core processor. In that model, each task is modeled with two properties, the required time to execute the task and a time interval. The semantics of the model is, that whenever the time interval of a task is over, an instance of the task is spawned that needs to be executed before the next instance of the same task is spawned. Thus, the problem of interest in this context is to determine whether it is possible to schedule the tasks such that each instance finishes in time.

Since this model was proposed, the design and structure of platforms that can be used to execute time-critical tasks has changed dramatically. Especially the rise of multiprocessor-systems and multi-core processors raises the requirement to parallelize the execution of single tasks. When modeling applications or tasks that are executed on a multi-core respectively -processor system, it is a common approach to model the application in terms of sub-tasks (also called jobs) in order to express the possibility of executing sub-tasks in parallel. In $P|prec|C_{\max}$ we are for example given a precedence constraint DAG of jobs in order to express which jobs can be executed simultaneously.

Besides this DAG-based model, that is for example proposed respectively used in [7, 10, 13, 40, 41, 51], several different models were proposed that also represent applications in terms of sub-tasks.

A first model is the so-called *fork/join model* that was proposed, respectively used, in [6, 36, 38]. The model represents a task as an alternating sequence of sequential and parallel segments, where the segments must be executed sequential and only the sub-tasks within a parallel segment can be executed simultaneously. Additionally, the model limits the number of sub-tasks per parallel segment and thus the number of sub-tasks that might be executed simultaneously.

The *synchronous parallel model*, see for example [3, 17, 44, 48, 53], is a generalization of the previous model. While the model does not limit the number of sub-tasks per parallel segment and allows consecutive parallel segments, it still enforces the segments to be executed sequential. The DAG-based model is the most general of the mentioned models.

The DAG-based model is often used in real-time scenarios. Baruah, Bonifaci, Marchetti-Spaccamela, Stougie, and Wiese [10] proposed the *sporadic DAG model*, where each recurrent task is represented by a DAG, a period and a deadline. Additionally, an execution time is given for each sub-task. A finite set of such sporadic DAG tasks then spawns a sequence of task instances. Whenever a task instance is spawned, all jobs of its DAG need to be executed within the deadline relative to the spawning. The period

denotes the minimum amount of time that passes between the spawning of two different instances of the same task. Given a finite set of such sporadic DAG tasks, one relevant problem is to determine whether each possible spawning sequence can be scheduled on a given number of processors such that all task instances finish within their respective deadline. This problem is also called a *schedulability test*. A number of publications consider the problem of schedulability testing sporadic DAG tasks, see for example [7, 12, 13, 27, 48, 49]. The NP-hardness of schedulability testing sporadic DAG tasks follows from classical scheduling results as listed in the following paragraph. As mentioned in the introduction, placing assumptions on the used scheduling algorithm is common in the context of schedulability testing. For example, [7], [48] and [49] assume that the jobs are scheduled according to a global *earliest deadline first (EDF)* rule, i.e., whenever a machine is free, the available job with the earliest deadline is started. Additionally, [27], [38] and [56] assume that the jobs are scheduled according to a fixed-priority order and [5, 34, 57] consider the computation of worst-case execution times for sporadic DAG tasks if fixed-priority scheduling is used. It is known, that this computation is NP-hard even on a single machine [12, 22], except for some special cases [14].

Additionally, directed acyclic graphs are also used to formulate precedence constraints between jobs in classical scheduling scenarios. The problem of scheduling jobs given in form of a precedence constraint DAG on a given number of machines with the objective to minimize the *makespan* ($P|prec|C_{\max}$) was shown to be strongly NP-hard by Ullman [58]. The makespan is the amount of time needed to finish all jobs. L. Graham [30, 37] introduced the list scheduling algorithm that is a 2-*approximation* for the problem. Garey and Johnson [28] showed that the problem is strongly NP-hard even if there are no precedence constraints between the given jobs. Lenstra, Kan, and Brucker [39] showed that the problem is weakly NP-hard if there are no precedence constraints and the number of machines is two. For non-empty precedence constraint graphs, Du, Leung, and Young [21] showed that the problem of minimizing the makespan is strongly NP-hard if the precedence constraint graph is a set of chains and the number of machines has a fixed value greater than one. Jansen and Solis-Oba [33] give a polynomial-time approximation scheme for the case where the precedence constraint graph is a fixed number of chains. Agnetis, Flamini, Nicosia, and Pacifici [1] showed that the problem of scheduling three chains on two machines to minimize the makespan is still weakly NP-hard and give a fully polynomial-time approximation scheme for the problem of scheduling a fixed number of chains on two machines. Finally, Hu [32] showed that the problem of scheduling trees with the objective to minimize the makespan can be solved in polynomial time if all jobs have the same processing time. As conditional DAGs are a generalization of non-conditional DAGs, the listed complexity results still hold for conditional variants of the problems, i.e., for problem variants that aim at deciding whether each possible execution of a conditional DAG can be scheduled within a given

deadline. However, while the problem of deciding whether the fixed-priority schedule for a given non-conditional scheduling instance and priority order completes within a given deadline is trivial, the complexity of the conditional variant to our knowledge remains open. Chapter 2 considers the complexity for this problem and exploits some of the listed results for non-conditional scheduling problems to derive hardness results. Note, that there is several work that considers scheduling problems with different objectives than the makespan. As the makespan is the only objective considered in this thesis, we do not list them at this point.

To allow the representation of control flow instructions, like for example `if-then-else` statements, several extensions of the DAG-based model were proposed. Besides the conditional DAG model, the so-called *multi-DAG* [26] was proposed, which models a task as a set of directed acyclic graphs that contains one DAG for each possible control flow. The semantics of the model is, that whenever a task is executed, exactly the sub-tasks of one of its DAGs need to be processed. The main problem that occurs when using this model to represent source code is, that a computer program might have an exponential number of control flows. Therefore, modeling existing programs with the multi-DAG model might not be feasible.

To overcome this problem, [9] respectively [46] introduced the model of *conditional DAGs*. In [45] and [46] several parameters are identified that characterize conditional DAG tasks. Furthermore, algorithms that compute the parameters and approximate schedulability tests for conditional DAGs in the sporadic setting are introduced. Sections 3.1.1 and 3.2.2 of this thesis will contemplate some of the algorithms and parameters when discussing algorithms to compute the worst-case execution time of a non-sporadic conditional DAG given a fixed-priority order. In addition, the chapter designs new algorithms specifically for the case of a single non-sporadic conditional DAG that is scheduled using a given fixed-priority order. In [9] a sufficient pseudo-polynomial schedulability test for conditional DAG tasks that are scheduled using EDF in the sporadic setting is proposed. Additionally, [8] introduces an algorithm for sporadic conditional DAG tasks using a different approach. Similar to the problem considered in this thesis, [11] proposes an approximate algorithm for a single conditional DAG task on a given number of machines. Again, the majority of this publications places assumptions on the used scheduling algorithm. While these publications recognize that schedulability testing conditional DAGs in the sporadic setting is NP-hard as implied by the previously listed results, the complexity of computing the worst-case execution time for a single non-sporadic conditional DAG using fixed priority scheduling to our knowledge remained open. Chapter 2 of this thesis considers the complexity of this case.

Besides the conditional DAG model, there are several other publications that consider similar models and corresponding problems. Chakraborty, Erlebach, and Thiele [16] consider a more restricted variant of the conditional DAG model, which models tasks as

a two-terminals DAG with the semantics that for each node exactly one successor needs to be executed. The model additionally annotates each edge with a delay such that a job can only be started if a predecessor has been completed and the corresponding delay has passed. Finally, each sub-task has an individual deadline relative to the point in time the sub-task becomes available to start. The paper then considers the problem of determining whether each possible combination of control flows for a given set of tasks can be scheduled such that each sub-task finishes within its deadline. They show that the preemptive version of the problem on a single machine is weakly NP-hard and give a fully polynomial-time approximation scheme. In [15] Chakraborty, Erlebach, Kunzli, and Thiele consider the non-preemptive uni-processor variant of the problem. They give an exact and sufficient condition for the schedulability under the assumption that the sub-tasks are scheduled using an earliest deadline first algorithm. They then derive a fully polynomial-time approximation scheme for the schedulability test. Their algorithm is an approximation in a sense that it might falsely determine instances to be schedulable. The approximation factor bounds the amount of time by which jobs in such instances can miss their deadline.

There a several models similar to the conditional DAG model that annotate each possible branch of a condition with the probability that the branch will be actually executed, see for example [24, 43]. The publications then consider the problem of optimizing several objectives in expectation, like for example the makespan.

Another conditional DAG-based model was proposed in [23]. In that model, non-conditional and conditional edges exist, where the latter are annotated with logical expressions. The semantics is that paths with conditional edges are only executed if the corresponding expressions evaluate to true. The paper considers the problem of computing a *scheduling table* that contains start times for the jobs dependent on the evaluation of the expressions.

Approaches to model the control flows of source code are also used in the context of software-engineering and, in specific, compilers. For example, Allen [2] defines *control flow graphs* in order to use them in a control flow analysis that enables compilers to execute source code optimizations. In contrast to conditional DAGs, control flow graphs are not necessarily acyclic and have a slightly different semantics. The special case of chain-like conditional DAGs, that is discussed in this thesis, is a special case of control flow graphs. Thus, there is some relation between control flow graphs and the content of this thesis.

1.3 Preliminaries

In the following, several preliminaries that are used during the course of this thesis are introduced. First, basic graph theoretical concepts and several types of graphs are de-

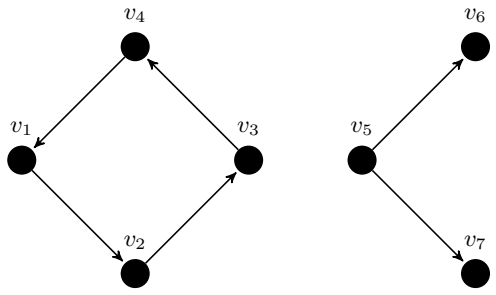
fined based on [19]. Then, scheduling basics and problems, like for example $P|prec|C_{\max}$, are formally established based on [60]. Subsequently, complexity theoretical preliminaries are introduced based on [4, 29] and finally, the conditional DAG model as well as corresponding problems are defined based on [9, 46].

1.3.1 Graph Theory

We start by introducing different basic graph theoretical concepts and properties.

Definition 1. A *directed graph* is a pair $G = (V, E)$ where V is a set of *vertices* (also called *nodes*) and $E \subseteq V \times V$ is a set of *edges*.

A directed graph $G = (V, E)$ is often visualized by illustrating each vertex $v \in V$ as a circle and each edge $(v, v') \in E$ as an arrow from v to v'. Figure 1.2 defines the example graphs G, G' and G'', and shows their visual representation.



$G = (V, E)$
- $V = \{v_1, v_2, v_3, v_4\}$
- $E = \{(v_1, v_2), (v_2, v_3), (v_3, v_4,)(v_4, v_1)\}$

$G' = (V', E')$
- $V' = \{v_5, v_6, v_7\}$
- $E' = \{(v_5, v_6), (v_5, v_7)\}$

$G'' = (V'', E'')$
- $V'' = V \cup V'$
- $E'' = E \cup E'$

Figure 1.2 Definition and visual representation of three example graphs G, G' and G''.

For an edge $e = (v, v')$ we say that e is an edge from v to v'. For a vertex $v \in V$ we say that $e \in E$ is an *incoming* edge of v if $e = (v', v)$ holds for some $v' \in V$. Analogous we say that e is an *outgoing* edge of v if $e = (v, v')$ holds for some $v' \in V$. Finally, e is called *incident* to v if e is an incoming or outgoing edge of v.

Consider again the example of Figure 1.2, then one can observe that each vertex and edge of example graph G respectively G' is also a vertex respectively edge of graph G''. Therefore, G and G' are *subgraphs* of G'' as defined in the following.

Definition 2. A graph $G = (V, E)$ is a *subgraph* of graph $G' = (V', E')$, written as $G \subseteq G'$, if $V \subseteq V'$ and $E \subseteq E'$ holds. G' is then called a *supergraph* of G.

The next definition introduces the notion of *paths*. According to the definition, the sequence of nodes $P = (v_1, v_2, v_3, v_4)$ in the example of Figure 1.2 is a path in G and G''.

Definition 3. A sequence $P = (v_0, v_1, \ldots, v_k)$ of nodes $v_i \in V$ of a directed graph

$G = (V, E)$ is a *path* from v_0 to v_k if $(v_i, v_{i+1}) \in E$ holds for each $i \in \{0, \ldots, k-1\}$. A path $P = (v_0, v_1, \ldots, v_k)$ is *simple* if $v_i \neq v_j$ holds for all $i, j \in \{0, \ldots, k\}$ with $i \neq j$. A path $P = (v_0, v_1, \ldots, v_k)$ is a *cycle* if $v_0 = v_k$ holds.

Given a path $P = (v_0, v_1, \ldots, v_k)$, we define $V(P) = \{v_0, \ldots, v_k\}$ as the set of nodes and $E(P) = \{(v_i, v_{i+1}) \mid i \in \{0, \ldots, k-1\}\}$ as the set of edges on P.

The sequence of nodes $P = (v_1, v_2, v_3, v_4)$ is a *simple* path in the example, as no vertex occurs more than once in the sequence. In contrast, $P' = (v_1, v_2, v_3, v_4, v_1)$ is a path in the example but not simple because v_1 occurs twice in P'. While P' is not simple, it is a cycle.

Definition 4. A graph $G = (V, E)$ is called *acyclic* if it does not contain any cycles.

As G and G'' contain the cycle P', they are not *acyclic*. On the other hand, G' does not contain any cycle and thus is acyclic.

The next definition introduces another graph property, the so-called *connectivity*. According to this definition, the example graphs G and G' are *connected* while G'' is not.

Definition 5. A directed graph $G = (V, E)$ is *connected* if for each $v, v' \in V$ with $v \neq v'$ a sequence $(v_0, \ldots, v_k)$ exists with $v_0 = v$, $v_k = v'$ and either $(v_i, v_{i+1}) \in E$ or $(v_{i+1}, v_i) \in E$ for each $i \in \{0, \ldots, k-1\}$.

Given a graph $G' = (V', E')$, a maximal connected subgraph G of G' is called *component* of G' as defined in the following definition.

Definition 6. A directed graph $G = (V, E)$ is a *component* of $G' = (V', E')$ if the following criteria hold:

- G is a subgraph of G'.
- G is connected.
- No connected graph $G'' \neq G$ with $G \subseteq G''$ and $G'' \subseteq G'$ exists.

According to this definition, G and G' are components of G'' in the example. Another property of a directed graph is its *width* as defined in the next definition.

Definition 7. Let $G = (V, E)$ be a directed graph and let $V' \subseteq V$ be a largest subset of V such that for each $v, v' \in V'$ with $v \neq v'$ neither a path from v to v' nor from v' to v exists. Then, $|V'|$ is the *width* of G.

Consider again the example of Figure 1.2, then the width of graph G is one, as there is a path between each pair of vertices in G. Additionally, the width of G' is two and the width of G'' is three. Using the previously introduced notions and concepts, the following paragraphs will define different types of graphs. The first definition defines *trees*.

Definition 8. A directed graph $G = (V, E)$ is a *tree* if it is either an *out-tree* or *in-tree*. A directed graph $G = (V, E)$ is an out-tree if it has a root $r \in V$ and for each $v \in V \setminus \{r\}$ there is exactly one path from r to v. A directed graph $G = (V, E)$ is an in-tree if it has a root $r \in V$ and for each $v \in V \setminus \{r\}$ there is exactly one path from v to r.

According to this definition, G' is a tree, in specific an out-tree, because v_5 is a root node and there is exactly one path from v_5 to any other node. In contrast, G is not a tree as it is a cycle and thus there are arbitrary many paths between any pair of nodes. Consequently, G'' is not a tree as well.

A special case of trees are *chains*, where a chain is a graph that consists of just one simple path as defined in the following.

Definition 9. A graph $G = (V, E)$ is a *chain* if it has the following form:

- $V = \{v_0, \ldots, v_k\}$
- $E = \{(v_i, v_{i+1}) \mid i \in \{0, \ldots k-1\}\}$

A graph $G = (V, E)$ is a *set of disjoint chains* if each component G' of G is a chain. The number of components of G then is the *number of chains* in G.

To conclude this section, the final paragraphs define *series-parallel graphs* according to [25].

Definition 10. A directed graph $G = (V, E)$ with terminals $s, t \in V$ is a *two-terminals series-parallel graph* if one of the following conditions holds:

- $V = \{s, t\}$ and $E = \{(s, t)\}$.
- G is the *series composition* of two two-terminals series-parallel graphs.
- G is the *parallel composition* of two two-terminals series-parallel graphs.

According to that definition, a two-terminal series-parallel graph is either a single edge or the composition of two smaller two-terminal series-parallel graphs. In the following, we define the two mentioned types of compositions while Figure 1.3 illustrates Definition 10.

Definition 11. The *parallel composition* of two two-terminal series-parallel graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with terminals $s_1, t_1 \in V_1$ and $s_2, t_2 \in V_2$ is the graph $P(G_1, G_2)$ that can be obtained by joining G_1 and G_2 and identifying $s = s_1 = s_2$ and $t = t_1 = t_2$.

Part *b*) of Figure 1.3 illustrates the parallel composition of two two-terminals graphs. Intuitively the parallel composition is the result of merging the sources respectively sinks of the two smaller two-terminals graphs.

Definition 12. The *series composition* of two two-terminals series-parallel graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with terminals $s_1, t_1 \in V_1$ and $s_2, t_2 \in V_2$ is the graph $S(G_1, G_2)$ that can be obtained by joining G_1 and G_2 and identifying $s = s_1$, $t = t_2$ and

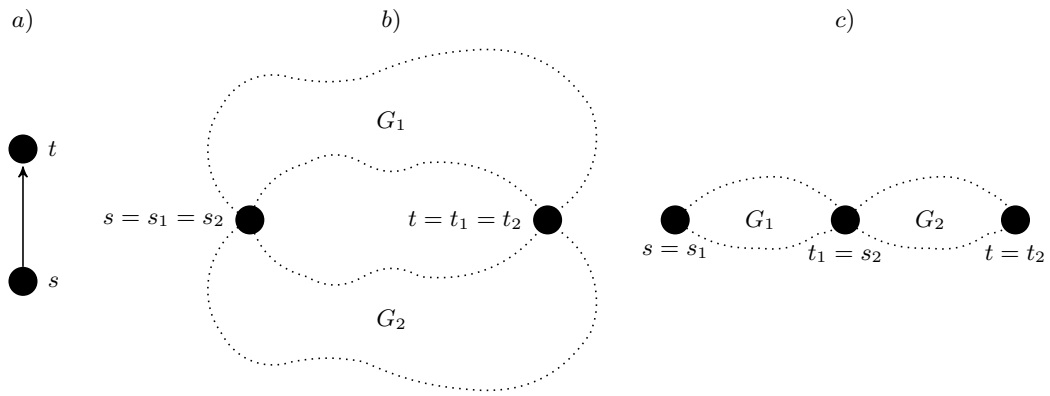


Figure 1.3 Illustration of the base case *a*) the parallel composition *b*) and the series composition *c*) of two-terminals series-parallel graphs.

$t_1 = s_2$.

Part *c*) of Figure 1.3 illustrates the series composition of two two-terminals graphs. Intuitively the series composition is the result of joining the target of one two-terminals graph with the source of another two-terminals graph.

1.3.2 Scheduling

This section introduces selected *scheduling problems* based on [60]. In all scheduling problems that are discussed during this thesis, we generally consider the problem of processing n jobs on m parallel identical machines. Each job $j \in \{1, \ldots, n\}$ needs to be executed by a machine for $p_j \in \mathbb{N}$ time units, where $p_j > 0$ is the *processing time* of job j. At that, each machine can process at most one job at a time and whenever a machine starts processing a job j, the execution cannot be interrupted until the p_j time units are over. We assume that the processing starts at point in time zero. In a *schedule* S for such a problem instance, each job is assigned to a *starting time* $S_j \in \mathbb{N}$ with $S_j \geq 0$ and executed from S_j up to its *completion time* $C_j = S_j + p_j$, i.e., during the interval $[S_j, C_j[$. The schedule is *feasible* if at each point in time $t \geq 0$ at most m jobs are being executed. Then, the completion time of the latest job to finish, $C_{\max} = \max_{j \in J} C_j$, denotes the *makespan* of the schedule, i.e., the point in time at which all jobs are finished. Figure 1.4 shows an example scheduling instance with two feasible schedules and their respective makespans. In the following, we will say that a machine is *idle* in schedule S at point in time t, if it does not process a job at t in S. In the example, machine m_1 is idle at point in time 3 in schedule S'. We call time intervals in which a machine is idle, *idle time*. Using this notations we can define the classical scheduling minimization problem $P||C_{\max}$.

Definition 13. In the makespan minimization problem $P||C_{\max}$ a set of n jobs with

$n = 5$
$m = 3$

$J = \{1, 2, 3, 4, 5\}$

Processing time	p_1	p_2	p_3	p_4	p_5
Value	2	2	3	3	5

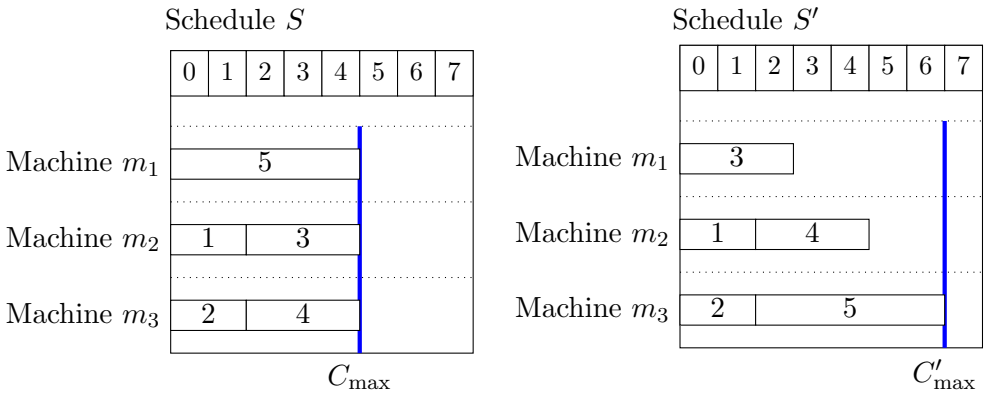


Figure 1.4 Example scheduling instance of $m = 3$ machines and $n = 5$ jobs $J = \{1, \ldots, 5\}$ with processing times p_j for each $j \in J$ and two feasible schedules S and S' with makespans $C_{\max}$ respectively $C'_{\max}$.

processing times $p_j > 0$ for each $j \in \{1, \ldots, n\}$ and a number of parallel identical machines m are given and the goal is to find a feasible schedule S with the minimum makespan.

The problem $P||C_{\max}$ is also called *load balancing* because each job j can be interpreted as an item with weight p_j. The n items then have to be distributed among the m machines, whereas each item has to be assigned to exactly one machine. The goal then is to minimize the maximum assigned weight over all machines. As all jobs have to be assigned to a machine, a total load of $\sum_{j=1}^{n} p_j$ has to be distributed among m machines. On average, each machine has a load of $\sum_{j=1}^{n} \frac{p_j}{m}$. Thus, for each schedule there must be at least one machine with a load of at least $\sum_{j=1}^{n} \frac{p_j}{m}$. Let $C^*_{\max}$ be the minimum makespan of an $P||C_{\max}$ instance, then we can conclude that the following inequality holds:

$$C^*_{\max} \geq \sum_{j=1}^{n} \frac{p_j}{m} \tag{1.1}$$

In a decision problem variant of $P||C_{\max}$ the goal is to decide whether the minimum makespan meets the lower bound of Equation (1.1). Garey and Johnson [28] showed that this decision problem is strongly NP-hard.

Definition 14. In the decision problem variant of $P||C_{\max}$ a set of n jobs with processing times $p_j > 0$ for each $j \in \{1, \ldots, n\}$ and a number of parallel identical machines m are given and the goal is to decide whether a feasible schedule S with a makespan of

$C_{\max} = \sum_{j=1}^{n} \frac{p_j}{m}$ exists.

In a more general variant of $P||C_{\max}$ not only a set of jobs $J = \{1, \ldots, n\}$ with processing times and a number of machines m are given, but also a directed acyclic *precedence constraint graph* $G = (V, E)$ with $V = J$. The problem is called $P|prec|C_{\max}$. A schedule for this problem is defined analogous to $P||C_{\max}$, but for a schedule S to be feasible, additionally $S_{j'} \geq C_j$ must hold for each $(j, j') \in E$. The following definition defines the optimization and decision problem variants of $P|prec|C_{\max}$, where the decision problem is defined slightly more general than for $P||C_{\max}$.

Definition 15. In $P|prec|C_{\max}$ a set of n jobs $J = \{1, \ldots, n\}$ with processing times $p_j > 0$ for each $j \in J$, a number of parallel identical machines m and a precedence constraint DAG $G = (V, E)$ with $V = J$ are given.

1. In the minimization problem variant the goal is to find the feasible schedule with the minimum makespan.
2. In the decision problem variant, we are additionally given a *deadline* $D \in \mathbb{N}$ and the goal is to decide whether a feasible schedule S with a makespan of $C_{\max} \leq D$ exists.

Figure 1.5 extends the example of Figure 1.4 with a precedence constraint graph to build an exemplary $P|prec|C_{\max}$ instance. According to the precedence constraint graph, jobs 2 and 3 can only be started after job 1 has been finished. Therefore both schedules in Figure 1.4 are not respecting the precedence constraints and thus not feasible for the $P|prec|C_{\max}$ instance. On the other hand, the schedule shown in Figure 1.5 is feasible for the example.

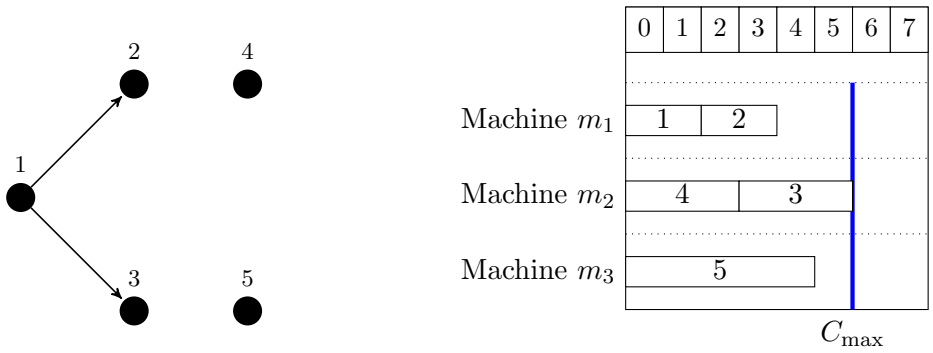


Figure 1.5 Example precedence constraints for the scheduling instance as defined in Figure 1.4 (left) and feasible schedule for the $P|prec|C_{\max}$ instance (right).

Note that $P|prec|C_{\max}$ is a generalization of $P||C_{\max}$ because each instance of the latter problem is an instance of $P|prec|C_{\max}$ with a precedence constraint graph $G = (V, E)$ with $E = \emptyset$. Thus, the strong NP-hardness of $P|prec|C_{\max}$ follows but was also shown by Ullman [58].

A simple algorithm for both problems is the so-called *list scheduling* or *fixed-priority scheduling*, which was introduced by L. Graham [30, 37]. In this algorithm, first some total order $\succ$ over all jobs in J is defined. Then, starting a point in time $t = 0$, whenever a machine is *free* and at least one job is *available*, the available job with the highest priority according to $\succ$ is started. A machine is free at a point in time t if no job is being processed on the machine at t. A job j is available at t if $C_{j'} \leq t$ holds for all $j' \in J$ with $(j', j) \in E$. While $P|prec|C_{\max}$ and $P||C_{\max}$ are defined to only have processing times greater than zero, we will consider problem variants that allow processing times of zero during the course of this thesis. To avoid ambiguities for such jobs j with $p_j = 0$, we say the successors of such jobs are available if all predecessors j with $p_j = 0$ have been started and all predecessors j with $p_j > 0$ have been completed. We will call schedules that are created using this algorithm *list schedules* or *fixed-priority schedules (FP-schedules)*.

Figure 1.6 shows a priority order $\succ$ for the example as well as the list schedule obtained using list scheduling with the order. At point in time zero all three machines are free and the three jobs 1, 4 and 5 are available. Jobs 2 and 3 are not available because of the precedence constraints. As only this three jobs are available, all of them are scheduled at point in time zero. At point in time two, another machines become free and the jobs 2 and 3 are available. Because 2 has a higher priority than 3 ($2 \succ 3$), job 2 is scheduled to start. Finally, at point in time three, the last job is started.

Priority Order: $1 \succ 2 \succ 3 \succ 5 \succ 4$

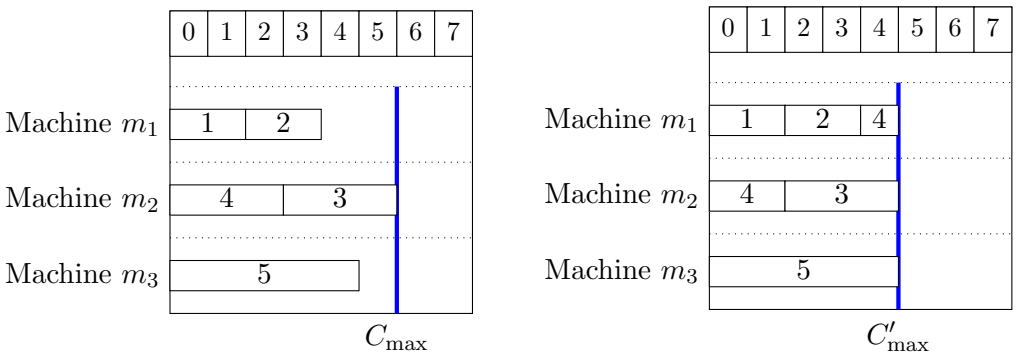


Figure 1.6 Priority order and list schedules for the example of Figures 1.4 and 1.5 for the case where preemption is allowed (right) and not allowed (left).

Note that the algorithm only needs to consider point in time zero and afterwards each point in time t where a scheduled job finishes. Therefore, the runtime of the algorithm is polynomial. Additionally, this algorithm computes reasonably good solutions. Let $C^*_{\max}$ be the minimum makespan of an $P|prec|C_{\max}$ instance and let $C_{\max}$ be the makespan of a list schedule on the same instance. Then, $C_{\max} \leq 2 \cdot C^*_{\max}$ holds. This performance guarantee was for example shown in [30, 37].

So far, we considered non-preemptive scheduling, i.e., once a job j is started, it needs to be processed until p_j time units are over and cannot be interrupted. In *preemptive scheduling*, a job j can be interrupted and the remaining processing time can be executed at a later point in time. In the preemptive version of $P|prec|C_{\max}$ for example, a variant of list scheduling is possible that at each point in time processes the (up-to) m available jobs with the highest priority. If a job j becomes available and all machines are busy processing other jobs, then a job j' with a lower priority than j can be interrupted and j can be started instead.

Figure 1.6 shows the preemptive list schedule for the example. The schedule starts analogous to the non-preemptive version, but at point in time two the jobs 2 and 3 become available and only one machine (m_1) is free. At that point, jobs 4 and 5 are being executed, but 2 and 3 have a higher priority. Per definition of preemptive list scheduling, the highest prioritized available jobs are always executed and thus job 4 is interrupted and continued later.

1.3.3 Complexity Theory

In the following, we will introduce some complexity theoretical basics that are used during the course of this thesis based on [4] and [29]. To define the necessary concepts, we need to use the notion of decision problems, which are problems whose outputs are either "Yes" or "No". In the previous section, we for example considered the decision problem variant of $P||C_{\max}$, where the output for a given instance is "Yes" if a schedule with $C_{\max} = \sum_{j=1}^{n} \frac{p_j}{m}$ exists and "No" otherwise. Another way of viewing decision problems is by considering the set of all problem instances $\{0,1\}^*$ and the set of all "Yes" instances $S \subseteq \{0,1\}^*$. Note, that the set of all instances is $\{0,1\}^*$ because we assume that all instance are encoded binary.

Then, the goal of the problem is to decide whether $x \in S$ holds for a given $x \in \{0,1\}^*$. In the following, we will represent a decision problem as the set of its "Yes" instances. For example, $P||C_{\max}$ would contain the binary encoding of all tuples (n, m, p) such that the n jobs with the processing times p_j for each $j \in \{1, \ldots, n\}$ can be scheduled on m machines with a makespan of $C_{\max} = \sum_{j=1}^{n} \frac{p_j}{m}$.

We will now use this representation of decision problems to define *complexity classes*. In general, a complexity class is a set of decision problems. The first complexity class relevant for this thesis is P, which contains all decision problems that are *efficiently solvable*.

Definition 16. A decision problem $S \in \{0,1\}^*$ is *efficiently solvable* if there exists a polynomial time algorithm A such that A outputs "Yes" for each $x \in \{0,1\}^*$ if and only if $x \in S$. Note that A must be polynomial in the input size, i.e., the number of bits in x. P is the set of all decision problems that are efficiently solvable.

The second complexity class that is relevant for this thesis is NP, which contains all decision problems that have *efficiently verifiable proof systems*.

Definition 17. Let $S \subseteq \{0,1\}^*$ be a decision problem. The relation $B \subseteq \{0,1\}^* \times \{0,1\}^*$ is a *proof system* for S if

1. $(x,b) \in B$ implies $x \in S$.
2. $x \in S$ implies that $(x,b) \in B$ holds for some $b \in \{0,1\}^*$.

If $(x,b) \in B$, then b is called a *certificate* for x.

Let $|x|$ denote the length of an $x \in \{0,1\}^*$, i.e., the number of bits in x.

Definition 18. A proof system $B \subseteq \{0,1\}^* \times \{0,1\}^*$ is *polynomial* if

1. there is a polynomial p, such that $|b| \leq p(|x|)$ holds for all $(x,b) \in B$.
2. $B \in \mathsf{P}$, i.e., there is a polynomial time algorithm that decides whether $(x,b) \in B$ holds for a given $x \in \{0,1\}^*$ and $b \in \{0,1\}^*$.

We say that a proof system can be efficiently verified if it is polynomial. Using this notion we can finally define the complexity class NP as follows.

Definition 19. The complexity class NP is the set of all decision problems for which a polynomial proof system exists.

The question how P and NP interconnect is one of the fundamental questions in the field of complexity theory. The relation $\mathsf{P} \subseteq \mathsf{NP}$ holds. For a decision problem $S \in \mathsf{P}$ the relation $B = \{(x,x) \mid x \in S\}$ is a proof system by definition. Additionally, the certificates are obviously of polynomial size and given a tuple (x,x) we can use the polynomial time algorithm that decides whether $x \in S$ holds to decide if $(x,x) \in B$ holds as well. The polynomial time algorithm must exist because of $S \in \mathsf{P}$.

The open question is whether $\mathsf{NP} \subseteq \mathsf{P}$ holds, i.e., whether each problem in NP can be solved in polynomial time. As this question remains open, we define more concepts to identify the "most difficult" problems in NP.

Definition 20. A problem S is *polynomial time reducible* to problem S', written $S \leq_p S'$, if a function $f : \{0,1\}^* \to \{0,1\}^*$ exists such that

1. f can be computed in polynomial time.
2. $x \in S$ if and only if $f(x) \in S'$.

A problem S is NP-*hard* if $S' \leq_p S$ holds for each $S' \in \mathsf{NP}$.
A problem S is NP-*complete* if it is NP-hard and $S \in \mathsf{NP}$ holds.

Polynomial time reductions are a tool to compare the "difficulty" of decision problems. If $S \leq_p S'$ holds and S' can be solved in polynomial time, then S can be solved in polynomial time as well by executing the polynomial time reduction and subsequently the polynomial time algorithm for problem S'. Therefore, S is at most as "difficult" as

S'. If consequently a problem S is NP-hard, then S is at least as "difficult" as every other problem in NP. Thus, the NP-complete problems are the hardest problems in NP. Furthermore, if we were able to show that an NP-hard problem can be solved in polynomial time, then all problems in NP could be solved in polynomial time as well.

As most researchers conjecture that $\mathsf{P} \neq \mathsf{NP}$ holds, showing that a problem S is NP-hard is an indicator that S can probably not be solved in polynomial time. Therefore, for a given problem, we are interested in either finding polynomial time algorithms or showing that the problem is NP-hard. To show that a problem S is NP-hard, it is sufficient to show that some known NP-hard problem S' is polynomial time reducible to S. Then, per transitivity of $\leq_p$, all problems in NP are polynomial time reducible to S.

Let $S \subseteq \{0,1\}^*$ be a decision problem, then we sometimes are interested in the complement $\bar{S} = \{0,1\}^* \setminus S$ of S. We can use this notation to define the following complexity classes:

$$\mathsf{CoNP} = \{S \subseteq \{0,1\}^* \mid \bar{S} \in \mathsf{NP}\}$$
$$\mathsf{CoP} = \{S \subseteq \{0,1\}^* \mid \bar{S} \in \mathsf{P}\}$$

Using this definitions, we can first observe that $\mathsf{P} = \mathsf{CoP}$ holds. This is the case as a polynomial time algorithm A that solves a problem S in P respectively CoP can also solve $\bar{S}$ by just negating the output of A. Analogous to NP-hardness and -completeness we can define the corresponding properties for the complexity class CoNP.

Definition 21. A problem S is CoNP-*hard* if $S' \leq_p S$ holds for each $S' \in \mathsf{CoNP}$.
A problem S is CoNP-*complete* if it is CoNP-hard and $S \in \mathsf{CoNP}$ holds.

Similar to NP-hardness, the CoNP-hardness of a problem S is an indicator that S cannot be solved in polynomial time unless $\mathsf{P} = \mathsf{NP}$. If S could be solved in polynomial time, then $\mathsf{CoNP} \subseteq \mathsf{P}$ follows by the CoNP-hardness of S. If then $S \in \mathsf{NP}$ holds for some S, then $\bar{S} \in \mathsf{CoNP}$ and thus $\bar{S} \in P$ follows. As $\mathsf{CoP} = \mathsf{P}$ holds, $\bar{S} \in \mathsf{P}$ and $S \in \mathsf{P}$ can be concluded. Thus, $\mathsf{NP} \subseteq \mathsf{P}$ and therefore $\mathsf{P} = \mathsf{NP}$ could be derived.

During the course of the thesis we will show a number of CoNP-hardness results to show that problems cannot be solved in polynomial time unless $\mathsf{P} = \mathsf{NP}$.

The following lemma formulates a useful property that will extensively be used during the course of this thesis.

Lemma 1. Let S be a decision problem. S is NP-hard if and only if $\overline{S}$ is CoNP-hard.

Proof. Let S be an arbitrary NP-hard problem, then S is NP-hard if and only if each $S' \in \mathsf{NP}$ can be reduced to S in polynomial time, i.e., $S' \leq_p S$. This is the case if and only if there is a function f that can be computed in polynomial time such that for each $x \in \{0,1\}^*$ it holds that $x \in S' \Leftrightarrow f(x) \in S$, which is equivalent to $x \notin S' \Leftrightarrow f(x) \notin S$. Now, $x \notin S' \Leftrightarrow f(x) \notin S$ in turn is equivalent to $x \in \overline{S'} \Leftrightarrow f(x) \in \overline{S}$. Thus, for each

$\overline{S'} \in \mathsf{CoNP}$ a polynomial time reduction f to $\overline{S}$ exists. Therefore $\overline{S}$ is CoNP-hard. The other direction can be proven analogous. □

According to the lemma, it suffices to show that problem S is NP-hard to prove that $\overline{S}$ is CoNP-hard. We will use this property to derive CoNP-hardness results in Chapter 2.

To conclude this section, we define *pseudo-polynomial time algorithms* as well as corresponding properties. Recap, that algorithms have a polynomial runtime if the runtime is polynomial in the input size, i.e., the number of bits needed to encode the problem.

Definition 22. An algorithm A has a *pseudo-polynomial* runtime if its runtime is polynomial in the numeric value of the input.

If for example a problem contains a numeric value k and an algorithm A needs k instructions to solve the problem, the runtime of A is $\mathcal{O}(2^{\log_2 k})$. As $\log_2 k$ is the number of bits needed to encode k, the runtime of A is exponential in the input size while it is polynomial in the numeric value of k. Using this notion, we define a stronger form of hardness as introduced in [28].

Definition 23. A problem S is

1. strongly NP-hard if the unary encoded version S_1 of S is NP-hard.
2. strongly CoNP-hard if the unary encoded version S_1 of S is CoNP-hard.
3. strongly NP-complete if it is strongly NP-hard and in NP.
4. strongly CoNP-complete if it is strongly CoNP-hard and in CoNP.

Observe that, if there was a pseudo-polynomial time algorithm A for a strongly NP- respectively CoNP-hard problem S, then A could solve the unary encoded version S_1 in polynomial time because the size of the unary encoding of a numeric value equals the numeric value. As S_1 is NP- respectively CoNP-hard by definition, this contradicts $\mathsf{P} \neq \mathsf{NP}$. Thus, a pseudo-polynomial time algorithm for a strongly NP-hard problem cannot exist, unless $\mathsf{NP} = \mathsf{P}$. This observation is formulated in [28].

Note, that Lemma 1 also works for strong NP- respectively CoNP- hardness for similar reasons than in the proof above.

Lemma 2. Let S be a decision problem. S is strongly NP-hard if and only if $\overline{S}$ is strongly CoNP-hard.

Proof. Let S be an arbitrary strongly NP-hard problem. Then, by Definition 23, the unary encoded version S_1 of S is NP-hard. It follows per Lemma 1, that $\overline{S_1}$ is CoNP-hard. As the unary encoded version $\overline{S_1}$ is CoNP-hard, it follows that $\overline{S}$ is strongly CoNP-hard. The other direction can be proven analogous. □

1.3.4 Conditional Directed Acyclic Graphs

In previous sections, various scheduling problems were considered, including problems where jobs in form of a directed acyclic graph are given, and the goal is to analyze their schedulability, whereas we assumed that all given jobs need to be executed.

However, in real-world scenarios there might occur situations where, dependent on the evaluation of some condition c, either a job j or a job j' needs to be executed but never both. If the evaluation of c is not known, but we still want to analyze the schedulability of jobs that might be executed, we need to consider all possible evaluations of all conditions. In order to do so, we need a model that allows to represent such conditions, which is not the case for directed acyclic graphs.

While we already saw an intuitive example for the representation of source code with *conditional DAGs* in the introduction, see Figure 1.1, the remainder of this section formally introduces the model based on the definitions given in [9] respectively [46].

Definition 24. A *conditional DAG* $G = (V, E, C)$ is a directed acyclic graph (V, E) and a set of *conditional pairs* $C \subseteq V \times V$ such that for each conditional pair $c_i = (v_i, \overline{v_i}) \in C$ the following holds:

1. There are exactly b_i outgoing edges from v to some nodes $s_{i1}, s_{i2}, \ldots, s_{ib_i}$ as well as exactly b_i incoming edges to $\overline{v_i}$ from some nodes $t_{i1}, t_{i2}, \ldots, t_{ib_i}$ for some $b_i \geq 2$. For each $l \in \{1, \ldots, b_i\}$ let P_{il} be the set of all paths from s_{il} to t_{il} in G. We define $G_{il} = (V_{il}, E_{il})$ as the union of all paths from s_{il} to t_{il}, i.e., $V_{il} = \bigcup_{p \in P_{il}} V(p)$ and $E_{il} = \bigcup_{p \in P_{il}} E(p)$.
   $V_{il} \neq \emptyset$ must hold for each $l \in \{1, \ldots, b_i\}$ and thus it follows that G_{il} is a DAG with the sole source s_{il} and the single sink t_{il}. In the following, we call the subgraphs G_{il} *conditional branches* and s_{il} respectively t_{il} the source and sink of G_{il}. Consequently, b_i is the number of branches.
2. It must hold that $V_{il} \cap V_{il'} = \emptyset$ for all l, l' with $l \neq l'$. Additionally, for all $l \in \{1, \ldots, b_i\}$ it must hold that $E \cap ((V \setminus V_{il}) \times V_{il}) = \{(v_i, s_{il})\}$ and $E \cap (V_{il} \times (V \setminus V_{il})) = \{(t_{il}, \overline{v_i})\}$. That is, there must not be edges from a conditional branch to nodes outside of that branch and vice versa except for the two edges incident to the start- and endpoint of the conditional pair.

For each $c_i = (v_i, \overline{v_i}) \in C$, the nodes v and $\overline{v_i}$ are called *conditional nodes* and v_i respectively $\overline{v_i}$ are the *start* and *end* of c_i.

The subgraph G_i that contains all branches of c_i, the nodes v_i and $\overline{v_i}$ and the edges that connect them, is called *condition*, i.e., $G_i = (V_i, E_i)$ with $V_i = \{v_i, \overline{v_i}\} \cup \bigcup_{l=1}^{b_i} V_{il}$ and $E_i = \bigcup_{l=1}^{b_i} E_{il} \cup \{(v_i, s_{il})\} \cup \{(t_{il}, \overline{v_i})\}$.

Note that [9] and [46] define conditional DAGs as graphs with a single source and sink. In this thesis, we will consider the general case where a sole source and sink do not necessary exist. If we interpret all non-conditional nodes of a conditional DAG

$G = (V, E, C)$ as jobs, the semantics of G is, that exactly the jobs of one conditional branch G_{il} per conditional pair $c_i = (v_i, \overline{v_i}) \in C$ need to be executed[1].

When visualizing conditional DAGs, the start and end of a conditional pair will be illustrated by squares whereas all other nodes are circles. Figure 1.7 shows an example of such a visualization. The depicted DAG qualifies for a conditional DAG, as the start of each conditional pair has as many outgoing edges as the end has incoming edges and no edges connect a conditional branch G_{il} with nodes outside of the branch, apart from (v_i, s_{il}) and $(t_{il}, \overline{v_i})$. Note, that nested conditional branches as used in the example are allowed by Definition 24 as well as parallelism inside of a conditional branch. Additionally, the picture illustrates that the source of a conditional branch can be the same node as its sink (see for example node 4) and that source respectively sink of a conditional branch can be start and end of another conditional pair (see for example nodes v_2 and $\overline{v_2}$).

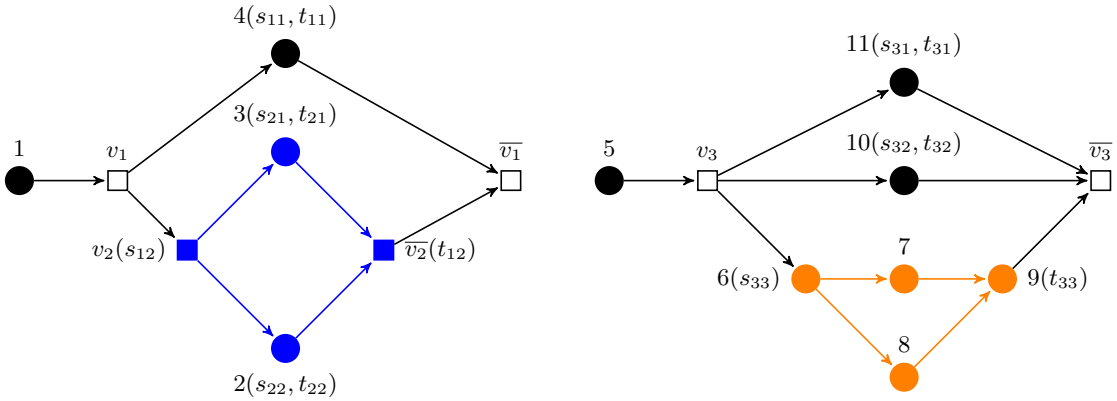


Figure 1.7 Example of a conditional DAG $G = (V, E, C)$. The blue colored subgraph is the condition G_2 of conditional pair $c_2 = (v_2, \overline{v_2})$, while the orange colored subgraph is one conditional branch of conditional pair $c_3 = (v_3, \overline{v_3})$.

From now on, we will interpret all non-conditional nodes as jobs and all conditional nodes as a tool to model conditions and different possible control flows. Each possible control flow then defines one possible non-conditional scheduling instance that could be chosen to be processed. In the following, we define *realization functions* in order to address one possible subgraph of a conditional DAG $G = (V, E, C)$ that could eventually be executed. Intuitively, a realization function maps each condition to the index of the conditional branch that is part of the subgraph to be executed.

Definition 25. A function $r : C \to \mathbb{N}_0$ is a *realization function* of a conditional DAG $G = (V, E, C)$ if the following holds:

1. $r(c_i) = 0$ if and only if G has a conditional branch $G_{i'l}$ such that $G_i \subseteq G_{i'l}$ and $r(c_i') \neq l$, i.e., $r(c_i) = 0$ signals that the conditional branches of c_i are nested into a

[1] Actually, this statement is only true if no conditional branches are nested into other conditional branches, as will be explained in the following.

conditional branch of another conditional pair that is not chosen to be processed.

2. $r(c_i) \leq b_i$, i.e., the conditional pair is mapped to a conditional branch that actually exists.

$\mathcal{R}_G$ is the set of all realization functions for conditional DAG G.

Therefore, given a conditional DAG $G = (V, E, C)$ and a realization function $r : C \to \mathbb{N}_0$, the semantics of r is, that a conditional branch G_{il} is chosen to be processed for r if and only if $r(c_i) = l$. A realization function can be used to formulate one possible combination of jobs that might be executed for a given conditional DAG.

Condition 1. of Definition 25 prevents conditional branches from being chosen to be processed if they are nested into conditional branches that are chosen to not be processed. Condition 2. on the other hand ensures that only conditional branches that are actually present in the conditional DAG can be chosen. Table 1.1 lists some example functions that either fulfill or violate the criteria of a realization function for the conditional DAG shown in Figure 1.7.

Function r	Valid/Invalid	Explanation
$r(c_1) = 1$, $r(c_2) = 0$, $r(c_3) = 3$	Valid	-
$r(c_1) = 2$, $r(c_2) = 1$, $r(c_3) = 3$	Valid	-
$r(c_1) = 1$, $r(c_2) = 1$, $r(c_3) = 3$	Invalid	$r(c_2) = 1$ violates Condition 1.
$r(c_1) = 2$, $r(c_2) = 1$, $r(c_3) = 0$	Invalid	$r(c_3) = 0$ violates Condition 1.
$r(c_1) = 2$, $r(c_2) = 1$, $r(c_3) = 4$	Invalid	$r(c_3) = 4$ violates Condition 2.

Table 1.1 Possible realization functions for the example in Figure 1.7 with explanations why they are valid realization functions or not.

In the following, we will call all jobs, conditional branches and paths *active* if they are chosen to be executed given a realization function.

Definition 26. Given a conditional DAG $G = (V, E, C)$ and a realization function $r : C \to \mathbb{N}_0$, a node $v \in V$ is *active* if one of the following conditions holds:

1. For all $c_i \in C$ and $l \in \{1, \ldots, b_i\}$ it holds that $v \notin V_{il}$, i.e., v is not part of any conditional branch and executed independent of the realization function.
2. Let $G_{il} = \{V_{il}, E_{il}\}$ be the innermost conditional branch[2] with $v \in V_{il}$, then $r(c_i) = l$ holds.

Let P be a path in G, then we will call P active if each node v on P is active. Additionally we will call a conditional branch G_{il} active if $r(c_i) = l$ holds.

Figure 1.8 exemplary marks jobs, paths and conditional branches that are active in the illustrated conditional DAG for a given realization function. Note, that jobs that are not part of any conditional branch always need to be executed and thus Condition 1. of

[2]Note that, as per Definition 24 no conditional branches of the same conditional pair share any nodes, the innermost conditional branch of a node is unique.

Definition 26 always qualifies them as active. Therefore, the nodes $1, 5, v_1, \overline{v_1}, v_3$ and $\overline{v_3}$ are marked as active in the figure. Additionally, being part of a conditional branch that is chosen to be processed is not enough for a job to be active because a job can be part of multiple nested conditional branches. Thus, Condition 2. of Definition 26 requires the innermost conditional branch that contains a job to be active. Therefore, node 3 is not marked as active in the example.

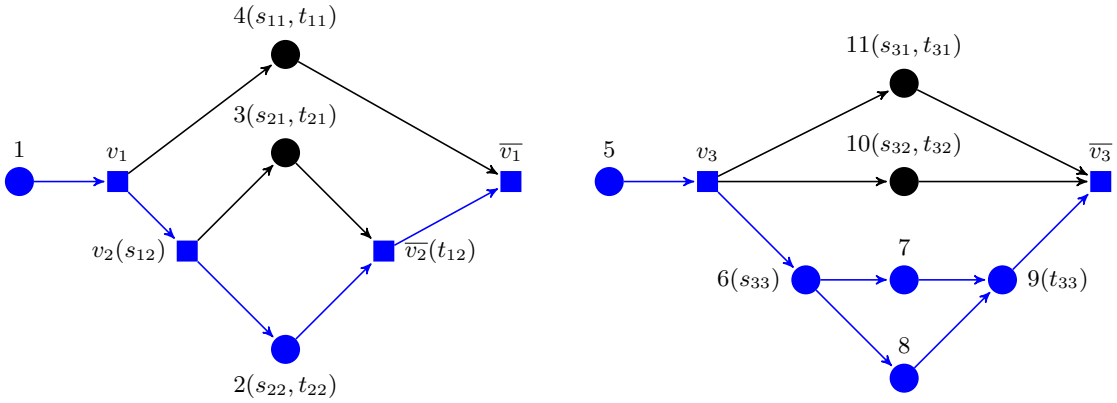


Figure 1.8 Example of a conditional DAG $G = (V, E, C)$. The blue colored nodes are active for realization function $r : C \to \mathbb{N}_0$ with $r(c_1) = 2$, $r(c_2) = 2$ and $r(c_3) = 3$

Let $G = (V, E, C)$ be a conditional DAG and r a realization function for G. Then r defines a subgraph of G that contains all nodes that are active for r as well as incident edges. In the following we will call such subgraphs *realizations* of G.

Definition 27. Given a conditional DAG $G = (V, E, C)$ and a realization function $r : C \to \mathbb{N}_0$, subgraph $G_r = (V_r, E_r)$ is a *realization* of G with:

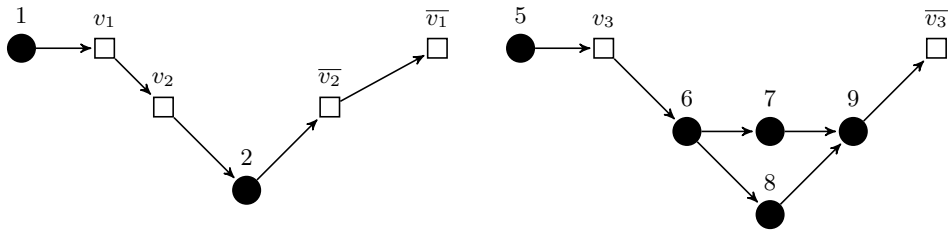
- V_r is the set of active nodes in G given r.
- Let I_r be the set of edges that are incident to nodes that are not active in G given r, then $E_r = E \setminus I_r$.

Figure 1.9 shows two example realizations for the conditional DAG $G = (V, E, C)$ as illustrated in Figure 1.7.

We now can generalize the previously introduced scheduling problems to work for conditional DAGs. Assume we are given a conditional DAG $G = (V, E, C)$, processing times p_i for each $i \in V$, a number of identical parallel machines m and a deadline D. Then, a realization G_r, the processing times p and the number of machines m define a non-conditional $P|prec|C_{\max}$ instance (G_r, p, m). In a conditional version of $P|prec|C_{\max}$ we ask the question whether for each $r \in \mathcal{R}_G$ the instance $I = (G_r, p, m)$ can be scheduled within the deadline, i.e., if there is a schedule S for I with a makespan $C_{\max} \leq D$.

This problem is clearly a generalization of $P|prec|C_{\max}$ as each instance of $P|prec|C_{\max}$ is also an instance of the conditional version that uses a conditional DAG $G = (V, E, C)$

1. Realization G_r with $r(c_1) = 2$, $r(c_2) = 2$ and $r(c_3) = 3$:



2. Realization $G_{r'}$ with $r'(c_1) = 1$, $r'(c_2) = 0$ and $r'(c_3) = 1$:

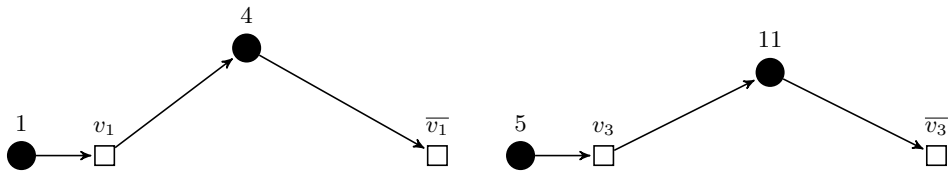


Figure 1.9 Two realizations for the example conditional DAG $G = (V, E, C)$ as shown in Figure 1.7.

with $C = \emptyset$. Therefore, the strong NP-hardness of the conditional version follows.

It seems to already be "difficult" to find a schedule that respects a given deadline, independent of whether conditions are used. Therefore, consider a variant of the problem and assume that for each realization G_r of conditional DAG $G = (V, E, C)$ it is known how G_r will be scheduled if it is chosen to be processed. Thus, we assume that we are given a fixed-priority order over all jobs in G and that each realization G_r is scheduled using list scheduling with this fixed-priority order. Note, that this assumption is common in the context of scheduling DAGs as outlined in Section 1.2.

During the rest of this thesis, we will only consider this problem variant and for the ease of notation define the input for this problem as a *conditional DAG task*.

Definition 28. A *conditional DAG task* T is a tuple $T = (G, p, \succ, m)$ with:

- $G = (V, E, C)$ is a conditional DAG.
- p is a vector of processing times, such that $p_i \in \mathbb{N}_0$ denotes the processing time of node $i \in V$. At that, $p_i = 0$ must hold for all conditional nodes $i \in V$.
- $\succ$ is the *fixed-priority order* of T, where $\succ$ is a total order over V such that $i \succ j$ denotes that i has a higher priority than j.
- m is a number of parallel identical machines.

Definition 29 defines the schedule of a realization G_r as the fixed-priority schedule on

$I = (G_r, p, m)$ that schedules the jobs of G_r according to the fixed-priority order $\succ$. As it is known how each realization will be scheduled, it is also known how long it takes to process the realization, i.e., the *execution time* of G_r is known.

Definition 29. Given a conditional DAG task $T = (G, p, \succ, m)$ and a realization $G_r = (V_r, E_r)$, the schedule $S_T(G_r)$ of realization G_r is defined as the fixed-priority schedule of the $P|prec|C_{\max}$ instance (G_r, p, m) that uses order $\succ$. The *execution time* $C_T(G_r)$ of a realization G_r given T is the makespan of $S_T(G_r)$.

Consider for example again the realizations of Figure 1.9, then Figure 1.10 shows the schedules and execution times of the realizations as defined by Definition 29.

Processing times:

Processing time	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}
Value	3	2	2	2	2	3	4	2	2	4	2

Fixed-priority order:

$$1 \succ 5 \succ v_1 \succ v_2 \succ v_3 \succ 2 \succ 3 \succ 4 \succ 6 \succ 7 \succ 8 \succ 9 \succ 10 \succ 1 \succ \overline{v_1} \succ \overline{v_2} \succ \overline{v_3}$$

Schedules:

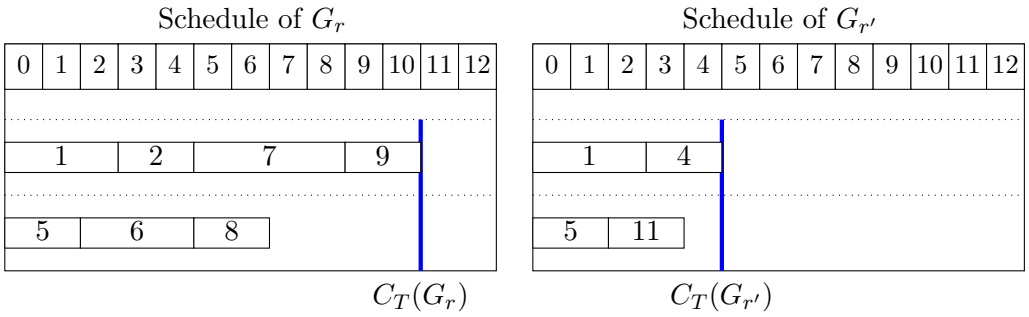


Figure 1.10 Schedules for the realizations as shown in Figure 1.9 on $m = 2$ machines using the depicted processing times and fixed priority order. Note, that conditional nodes per definition have a processing time of zero and thus are neither listed in the processing time table nor shown in the schedules.

Using this notations we can also define the *worst-case execution time* $WCET(T)$ of a conditional DAG task $T = (G, p, \succ, m)$ as the maximum execution time over all realizations of G as formulated in the following definition.

Definition 30. Given a conditional DAG task T, the *worst-case execution time* $WCET(T)$ of T is defined as

$$WCET(T) = \max_{r \in \mathcal{R}_G} C_T(G_r)$$

Consider the problem where a conditional DAG task $T = (G, p, \succ, m)$ and a deadline D are given and assume that each realization of G is scheduled as defined above. Then,

the goal is to decide whether $C_T(G_r) \leq D$ holds for all $r \in \mathcal{R}_G$, which is the case if and only if $WCET(T) \leq D$ holds. Thus, if we can calculate the worst-case execution time of T, we can decide whether each realization of G can be completed within D. Therefore, the computational problem of computing the worst-case execution time for a given conditional DAG task is a problem of interest in the context of conditional DAGs.

Definition 31. In the *worst-case execution time problem* for conditional DAG tasks, a conditional DAG task T is given and the goal is to compute $WCET(T)$.

In contrast to the conditional variant of $P|prec|C_{\max}$ where no fixed-priority order is given, the worst-case execution time problem is trivial if the given conditional DAG does not use conditions. Given $T = (G, p, \succ, m)$ such that G does not use conditions, G only has a single realization and therefore the worst-case execution time for the fixed-priority order $\succ$ can be easily computed.

The chapters 2 and 3 consider the complexity of the worst-case execution time problem for conditional DAG tasks and derive algorithms to compute or approximate the worst-case execution time for a given conditional DAG task T.

Chapter 2

Complexity

In this chapter, we investigate the complexity of the worst-case execution time problem for conditional DAGs given a fixed-priority order. To do so, the first section of this chapter defines two decision problem variants of the computational worst-case execution time problem.

While Definition 24 defined conditional branches to not have any common nodes and excluded edges between different conditional branches, the first section of this chapter considers a slightly more general model. We show that the two introduced decision problems are strongly NP- respectively CoNP-hard for the more general model even if the task is executed on a single machine. This is in contrast to the originally defined problem, which can be solved in polynomial time on a single machine, see [46] and Section 3.1.1.

Afterwards, the chapter discusses the complexity of the worst-case execution time problem for conditional DAG tasks as originally defined in Section 1.3.4. After the hardness of the general case is shown, the final sections of this chapter consider the hardness for more restricted classes of conditional DAGs. We will see that the worst-case execution time problem remains strongly NP- respectively CoNP-hard even if we only consider two-terminals series-parallel conditional DAGs or allow preemption.

Furthermore, Section 1.2 discussed that non-conditional $P|prec|C_{\max}$ is NP-hard and remains NP-hard if the precedence constraint graph is restricted to be a constant number of chains or a tree. In the later sections of this chapter, similar restrictions to conditional DAGs are discussed in the context of the worst-case execution time problem.

While the usage of conditional pairs prevents a conditional DAG from being a tree or a set of disjoint chains, conditional DAGs such that each realization is a tree respectively a set of disjoint chains are possible. Therefore, the final sections of this chapter consider those cases and show that the problem remains strongly CoNP-hard if each realization is a tree and is weakly CoNP-hard if each realization is a constant number of disjoint chains. During the course of this chapter we will also derive some hardness results for cases where the number of machines is fixed.

2.1 Preliminaries

In order to discuss the complexity of the worst-case execution time problem for a conditional DAG given a fixed-priority order, its decision problem variant needs to be considered. In a natural decision problem variant, a conditional DAG task $T = (G, p, \succ, m)$ and a deadline $D \in \mathbb{N}$ are given and the goal is to decide whether $C_T(G_r) \leq D$ holds for all $r \in \mathcal{R}_G$. This is the case if and only if $WCET(T) \leq D$ holds. In terms of formal languages this decision problem can be represented as follows, where $\mathcal{T}$ is the set of all conditional DAG tasks:

$$\begin{aligned}\mathsf{C\text{-}DAG} &= \{(T = (G, p, \succ, m), D) \subseteq \mathcal{T} \times \mathbb{N} \mid \forall r \in \mathcal{R}_G : C_T(G_r) \leq D\} \\ &= \{(T = (G, p, \succ, m), D) \subseteq \mathcal{T} \times \mathbb{N} \mid WCET(T) \leq D\}\end{aligned}$$

A first step towards classifying C-DAG in terms of complexity is to determine which complexity classes contain the problem. Let us therefore start by arguing whether C-DAG $\in$ NP holds. Unfortunately, it is not obvious whether polynomial proof systems for C-DAG exist. Intuitively, the $\forall$-quantifier in the definition of C-DAG seems to make the verification of hypothetical polynomial certificates rather "difficult".

Therefore, consider the complement of C-DAG, which is the decision problem where a conditional DAG task $T = (G, p, \succ, m)$ and a deadline D are given and the goal is to decide whether there exists an $r \in \mathcal{R}_G$ such that $C_T(G_r) > D$ holds. This is the case if and only if $WCET(T) > D$ holds. In terms of formal languages, the complement can be represented as follows:

$$\begin{aligned}\overline{\mathsf{C\text{-}DAG}} &= \{(T = (G, p, \succ, m), D) \subseteq \mathcal{T} \times \mathbb{N} \mid \exists r \in \mathcal{R}_G : C_T(G_r) > D\} \\ &= \{(T = (G, p, \succ, m), D) \subseteq \mathcal{T} \times \mathbb{N} \mid WCET(T) > D\}\end{aligned}$$

Now, when considering the complement, we can show that $\overline{\mathsf{C\text{-}DAG}} \in$ NP holds as formulated in Theorem 1.

Theorem 1. The problem of deciding whether there exists a realization G_r with $C_T(G_r) > D$ for a given conditional DAG task $T = (G, p, \succ, m)$ and a deadline D is in NP, i.e., $\overline{\mathsf{C\text{-}DAG}} \in$ NP.

Proof. In order to show the theorem, it is sufficient to give a polynomial proof system for $\overline{\mathsf{C\text{-}DAG}}$. Therefore consider the following relation:

$$B = \{((T = (G, p, \succ, m), D), G_r) \mid T \in \mathcal{T}, D \in \mathbb{N}, r \in \mathcal{R}_G \text{ such that } C_T(G_r) > D\}$$

The remainder of this proof shows that B is a polynomial proof system for $\overline{\mathsf{C\text{-}DAG}}$. First, observe that G_r can be represented in polynomial space as it is a subgraph of

the given conditional DAG. Furthermore, given $T = (G, p, \succ, m)$, D and G_r, it can be decided in polynomial time whether $C_T(G_r) > D$ holds. To do so, we just need to construct schedule $S_T(G_r)$ for the non-conditional scheduling instance (G_r, p, m) to compute $C_T(G_r)$ and then verify $C_T(G_r) > D$. As $S_T(G_r)$ can be computed using a list scheduling algorithm, it can be done in polynomial time.

Finally, it remains to show that $((T, D), G_r) \in B$ implies $(T, D) \in \overline{\text{C-DAG}}$ and that, if $(T, D) \in \overline{\text{C-DAG}}$ holds, there is an G_r such that $((T, D), G_r) \in B$ holds.

Therefore, consider a tuple $t = ((T, D), G_r) \in B$, then $t \in B$ holds only if $C_T(G_r) > D$ holds, which is the case only if $(T, D) \in \overline{\text{C-DAG}}$. Finally, if $(T, D) \in \overline{\text{C-DAG}}$, then per definition there exists an $r \in \mathcal{R}_G$ with $C_T(G_r) > D$ and thus $((T, D), G_r) \in B$. □

Using $\overline{\text{C-DAG}} \in \text{NP}$, it follows by definition of CoNP that C-DAG $\in$ CoNP holds as formulated in Theorem 2.

Theorem 2. The problem of deciding whether $C_T(G_r) \leq D$ holds for all $r \in \mathcal{R}_G$ given a conditional DAG task $T = (G, p, \succ, m)$ and a deadline D is in CoNP, i.e., C-DAG $\in$ CoNP.

In the following sections, $\overline{\text{C-DAG}}$ will be considered and shown to be NP-hard – and thus NP-complete – in different variations. According to Lemma 1, by showing that $\overline{\text{C-DAG}}$ is NP-hard, we automatically show that C-DAG is CoNP-hard and thus CoNP-complete. In the next sections, we derive NP-hardness and thus CoNP-hardness results.

2.2 Shared-Node Model

In this section, we consider a variant of conditional DAGs where edges between conditional branches of different conditions are allowed. A *conditional DAG with shared nodes* $G = (V, E, C)$ is defined analogous to Definition 24 with an adjusted second requirement for each $(v_i, \overline{v_i}) \in C$ that allows edges from conditional branches G_{il} to conditional branches $G_{jl'}$ of pairs $(v_j, \overline{v_j}) \in C$ with $(v_i, \overline{v_i}) \neq (v_j, \overline{v_j})$. Figure 2.1 shows an exemplary conditional DAG with shared nodes.

In order to compensate for the different model, the definition of activity has to be adjusted because the notion of the innermost conditional branch that contains a node v, as used in Definition 26, is not unambiguous anymore. Therefore, we adjust Condition 2. of Definition 26 and say that a node v is active, if at *least one* of the innermost conditional branches G_{il} with $v \in V_{il}$ is active. In the conditional DAG of Figure 2.1 it would for example be enough for v_{shared} to be active if the lower branch of the upper condition or the upper branch of the lower condition is active. Thus, v_{shared} is executed if at least one of those branches is executed. All other definitions remain the same as for the original variant of conditional DAGs.

In Section 3.1.1 we will see a polynomial time algorithm for the single machine version of the worst-case execution time problem for conditional DAGs as originally defined in

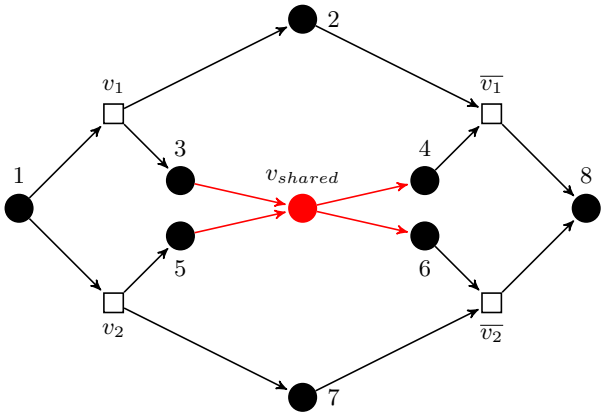


Figure 2.1 Example of a conditional DAG with shared nodes. Nodes and edges that violate the original definition of a conditional DAG are colored in red.

Definition 24. For the remainder of this section, we will show that the worst-case execution time problem for the introduced variant is indeed NP-hard even if the given task is scheduled on a single machine. Therefore, the following problem will be considered.

Definition 32. In the $\overline{\text{Shared C-DAG}}$ problem, a conditional DAG task $T = (G, p, \succ, 1)$ that allows G to have shared nodes and a deadline D are given and the goal is to decide whether there exists a realization G_r with $C_T(G_r) > D$.

In order to show the strong NP-hardness of $\overline{\text{Shared C-DAG}}$, a polynomial reduction from the 1in3-SAT-problem will be given. The 1in3-SAT-problem, which is defined in Definition 33, was shown to be strongly NP-hard by Schaefer [54].

Definition 33. In the 1in3-SAT problem a set of propositional logic 3-Clauses $C = \{C_1, \ldots, C_n\}$ and a set of variables $L = \{l_1, \ldots, l_n\}$ are given, such that each clause only contains positive literals and each variable occurs in exactly three clauses. The goal of the problem is to decide whether a satisfying variable assignment exists such that for each $C_i = \{l_{i1}, l_{i2}, l_{i3}\}$ exactly one l_{ij} is satisfied.

We will reduce an 1in3-SAT instance to a conditional DAG task $T = (G, p, \succ, 1)$ such that G allows shared nodes and has a single source and sink. Therefore, we will show the following theorem.

Theorem 3. $\overline{\text{Shared C-DAG}}$ is strongly NP-hard, even if the given conditional DAG has a single source and sink.

The following proof is based on previous work and ideas by Alberto Marchetti-Spaccamela, Nicole Megow, Martin Skutella and Leen Stougie.

Proof. Assume we are given a 1in3-SAT instance (C, L) as defined above. A conditional DAG task $T = (G, p, \succ, 1)$ that allows shared nodes and a deadline D can be constructed

as follows:

1. Let s and t be the single source and sink of G with processing times of zero.
2. For each clause $C_j \in C$, add a node c_j with a processing time of one.
3. For each variable $l_i \in L$, introduce two nodes v_i and $\overline{v_i}$ that form a conditional pair $cp_i = (v_i, \overline{v_i})$.
   (a) For each branch $l \in \{1, 2\}$ of this conditional pair, add a source s_{il} and a sink t_{il} with processing times of zero.
   (b) Add edges from s_{i1} to all nodes c_j with $l_i \in C_j$ and from each c_j with $l_i \in C_j$ to t_{i1}. Note that l_i occurs in exactly three clauses and therefore edges to three nodes are added.
   (c) Add two nodes p_i and q_i with processing times of one and edges from s_{i2} to p_i and q_i as well as from p_i and q_i to t_{i2}.
4. Connect each conditional pair $cp_i = (v_i, \overline{v_i})$ that was introduced in the previous step to the source and sink of G by adding edges (s, v_i) and $(\overline{v_i}, t)$.
5. Use some arbitrary fixed-priority order $\succ$[1].
6. Set $D = \lceil \frac{7n}{3} \rceil - 1$.

Figure 2.2 illustrates the construction with an example of the constructed conditional pairs for two variables that share one clause C_l.

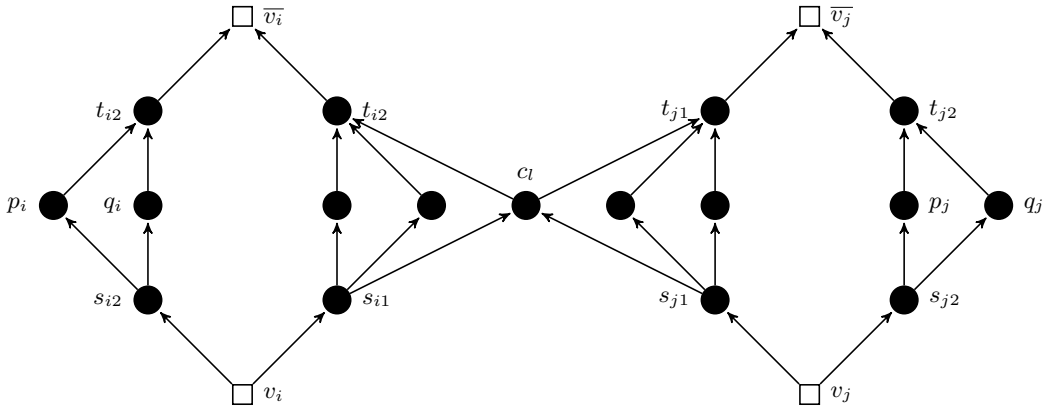


Figure 2.2 Example excerpt of the 1in3-SAT construction for two variables l_i and l_j that share one clause C_l.

To show that the given construction is indeed a polynomial reduction from 1in3-SAT to the worst-case execution time problem, observe that the constructed graph is a conditional DAG with shared nodes and has a single source and sink.

Next, we argue that the construction is of polynomial size. For each variable $l_i \in L$ the two conditional nodes v_i and $\overline{v_i}$ as well as the sources and sinks for both conditional branches, s_{i1}, s_{i2}, t_{i1} and t_{i2}, are introduced in addition to the nodes p_i and q_i. Therefore, $8n$ vertices are introduced over all variables. Additionally, for each clause C_j exactly

[1] As the constructed task is scheduled on a single machine, the fixed-priority order does not influence its worst-case execution time.

one node c_j is added. Finally, source s and sink t are added. All in all this leads to a constructed conditional DAG with $9n + 2$ nodes.

For each variable $l_i \in L$, exactly 8 respectively 6 edges are introduced for the conditional branches G_{i1} and G_{i2} in addition to two edges that connect the condition cp_i with s and t. All in all, $16n$ edges are added.

It follows that the constructed graph is of polynomial size and can be constructed in polynomial time.

Thus, to show that the construction is indeed a polynomial time reduction, it remains to prove its correctness and completeness. Therefore, we continue by proving the following two statements.

1. If the given 1in3-SAT instance has a feasible solution, the constructed graph has a worst-case execution time of at least $\frac{7n}{3}$.
2. If the given 1in3-SAT instance does not have a feasible solution, the constructed graph has a worst-case execution time of strictly less than $\frac{7n}{3}$.

If the two statements hold, correctness and completeness of the reduction follow. To see this, consider an instance (C, L) of the 1in3-SAT problem. Then, the instance is satisfiable by exactly one variable per clause if and only if $WCET(T) \geq \frac{7n}{3} > \lceil \frac{7n}{3} \rceil - 1 = D$ holds for the constructed instance (T, D) according to the two statements. Now, $WCET(T) > D$ holds if and only if $(T, D) \in \overline{\text{Shared C-DAG}}$. Therefore, the reduction is correct and complete if the two statements hold and the remainder of the proof will show that this is indeed the case.

Before we show the statements, observe that the execution time $C_T(G_r)$ of a realization G_r on a single machine is just the sum of processing times of all active jobs in G_r because all jobs need to be processed on a single machine and no idle time can occur by definition of schedule $S_T(G_r)$.

To prove the first statement, assume that a 1in3-SAT-instance that has a feasible solution is given. Then, a satisfying variable assignment $\alpha : L \to \{0, 1\}$ exists such that α satisfies each clause in C by exactly one literal. Using α, we can define the realization function $r : C \to \mathbb{N}_0$ for the constructed instance as follows.

$$r(cp_i) = \begin{cases} 1 & \text{if } \alpha(l_i) = 1 \\ 2 & \text{otherwise} \end{cases}$$

We will now argue that the execution time of realization G_r, and thus the worst-case execution time of T, is at least $\frac{7n}{3}$.

For each $l_i \in L$ the branch G_{i1} is active for r if and only if $\alpha(l_i) = 1$. In the resulting schedule, each c_j must be executed as α satisfies all clauses and thus for each clause C_j there is a literal $l_i \in C_j$ with $\alpha(l_i) = 1$ and thus $r(cp_i) = 1$. Therefore G_{i1} is active and thus $c_j \in V_{i1}$ is active as well. This contributes n time units to the execution time $C_T(G_r)$.

Because α satisfies each clause by exactly one literal and each variable occurs only as a positive literal in exactly three clauses, it follows by pigeon hole principle that at least $\frac{2n}{3}$ variables $l_i \in L$ exist with $\alpha(l_i) = 0$ and thus $r(cp_i) = 2$. Therefore, for each such l_i the branch G_{i2} is active and thus p_i and q_i are active as well. This contributes $\frac{4n}{3}$ time units to the execution time $C_T(G_r)$. Adding up both parts leads to an execution time of at least $\frac{4n}{3} + n = \frac{7n}{3}$.

To prove the second statement, observe that there is an one-to-one correspondence between variable assignments for the 1in3-SAT-instance and the activity of conditional branches in the constructed conditional DAG. This holds as for every variable assignment α the realization function $r : C \to \mathbb{N}_0$ can be constructed as described above. Per construction of r, for each different α a unique r is created. Furthermore, definition and value range of α and r are of equal size[2]. Therefore, by considering all possible variable assignments we consider all possible realization functions and thus all possible realizations and execution times.

Now, assume that the formula is not satisfiable by exactly one literal per clause and consider an arbitrary variable assignment α. Let k be the number of variables l_i with $\alpha(l_i) = 0$.

If $k < \frac{2n}{3}$, then at most n nodes c_j are active for the corresponding realization function. For k variables l_i, it holds that q_i and p_i are active for the corresponding realization function. Therefore, the execution time of realization G_r is at most $n+2k < n+\frac{4n}{3} = \frac{7n}{3}$.

If $k > \frac{2n}{3}$, then at most $3 \cdot (n-k)$ nodes c_j with $j \in \{1, \ldots, n\}$ are active for the corresponding realization function as each variable occurs in at most 3 clauses. Additionally for k variables l_i it again holds that q_i and p_i are active for the corresponding realization function. Therefore, the execution time of realization G_r is at most $2k + 3(n-k) < 2\frac{2n}{3} + 3(n - \frac{2n}{3}) = \frac{7n}{3}$.

To finish the proof, consider $k = \frac{2n}{3}$. As α does not satisfy the given formula by assumption, it follows from the pigeon hole principle that of the $n-k$ positive assigned variables at least two must occur in the same clause. This means that at least one node c_j is not active for the corresponding realization function. Therefore, the execution time of realization G_r is strictly less than $n + 2k = n + \frac{4n}{3} = \frac{7n}{3}$.

As this three cases for k cover all possible values, it follows that for each assignment α the corresponding realization function r has an execution time of strictly less then $\frac{7n}{3}$. Because by considering each α each realization function r is considered, $C_T(G_r) < \frac{7n}{3}$ holds for each realization function r. Thus, $WCET(T) < \frac{7n}{3}$ holds and the statement and thus theorem follow. □

The theorem states that $\overline{\text{Shared C-DAG}}$ is indeed strongly NP-hard. As $\overline{\text{Shared C-DAG}} \in$

[2]The value range of r formally is $\mathbb{N}_0$, but, by definition of realization functions, $r(c_i) \leq b_i$ must hold for each conditional pair c_i. Because each constructed pair has two branches and no pairs are nested into each other, c_i can effectively only be mapped to 1 or 2.

NP can be shown analogous to the proof of Theorem 1, i.e., by using the realization G_r with $C_T(G_r) > D$ as certificate, the following theorem can be deduced.

Theorem 4. $\overline{\text{Shared C-DAG}}$ is strongly NP-complete, even if the given conditional DAG has a single source and sink.

Additionally, per Lemma 1 and definition of CoNP, the strong NP-completeness of $\overline{\text{Shared C-DAG}}$ directly implies the following theorem.

Theorem 5. Shared C-DAG is strongly CoNP-complete, even if the given conditional DAG has a single source and sink.

2.3 General Case

In the previous section, the worst-case execution time problem was shown to be NP-hard for conditional DAGs that allow shared nodes between different conditional branches even if the DAG is executed on a single machine. As that problem is NP-hard on a single machine, it directly follows that it remains NP-hard if the DAG is executed on a number of parallel identical machines m that is part of the input.

While this is the case for the model that allows shared nodes, Section 3.1.1 will give a polynomial time algorithm for the single machine variant of the worst-case execution time problem for conditional DAGs as originally defined in Definition 24. Therefore, the complexity of C-DAG respectively $\overline{\text{C-DAG}}$ still needs to be discussed. The following subsection will show the strong NP-hardness of $\overline{\text{C-DAG}}$ and thus the strong CoNP-hardness of C-DAG.

Afterwards, the consecutive subsections will show different properties of the reduction that is used to prove the NP-hardness. In specific, we show that the reduction still works if only series-parallel conditional DAGs are considered and if preemption is allowed.

2.3.1 Strong NP-Hardness

In this subsection we show the strong NP-hardness of $\overline{\text{C-DAG}}$. To do so, the list scheduling makespan maximization problem (LS MAX) as defined in Definition 34 is shown to be strongly NP-hard. Subsequently, we show the NP-hardness of the worst-case execution time problem via reduction from LS MAX. The proof is partially based on previous work and ideas by Alberto Marchetti-Spaccamela, Nicole Megow, Martin Skutella and Leen Stougie.

Definition 34. In the *list scheduling makespan maximization problem* (LS MAX) we are given a precedence constraint DAG $G = (V, E)$, processing times $p_j \in \mathbb{N}$ with $p_j > 0$ for each $j \in V$, a number of identical parallel machines m and a deadline $D \in \mathbb{N}$. The goal of the problem is to decide whether the maximum makespan $\overline{C}_{\max}$ that can be achieved

using a list scheduling rule on the given instance is strictly greater than D.

We will now show that LS MAX is strongly NP-hard by giving a polynomial time reduction from $P||C_{\max}$, which is known to be strongly NP-hard as shown in [28].

Theorem 6. LS MAX is strongly NP-hard, even if no precedence constraints are used.

Proof. Let $J = \{1, \ldots, n\}$ be the set of jobs and $p_j > 0$ be the processing time for each $j \in \{1, \ldots, n\}$ for a given instance of $P||C_{\max}$ on m machines.

We construct an instance of LS MAX by using the same number of machines (m) as in the given instance and using the set of jobs $J \cup \{n+1\}$ where each $j \in J$ has the processing time p_j as in the given instance and $p_{n+1} = \sum_{j=1}^{n} \frac{p_j}{m}$ holds. That is, the processing time of $n+1$ equals the average load of the given instance. We construct the precedence constraint graph $G = (V, E)$ with $V = J \cup \{n+1\}$ and $E = \emptyset$ and use $D = \left(2 \cdot \sum_{i=1}^{n} \frac{p_i}{m}\right) - 1$ as deadline. Note that this construction can be executed in polynomial time and space.

To proof correction and completeness, assume that none of the original jobs has a processing time larger than the average load. This can be done without loss of generality because if there was a job j with $p_j > \sum_{j=1}^{n} \frac{p_j}{m}$, the instance trivially cannot be scheduled to achieve a makespan of $\sum_{j=1}^{n} \frac{p_j}{m}$. Thus, excluding such jobs does not affect the strong NP-hardness of $P||C_{\max}$.

To show the correctness and completeness of the reduction, the remainder of the proof shows that the maximum makespan of the constructed instance is $\overline{C}_{\max} = 2 \cdot \sum_{j=1}^{n} \frac{p_j}{m}$ if and only if the minimum makespan of the given $P||C_{\max}$ instance is $C^*_{\max} = \sum_{j=1}^{n} \frac{p_j}{m}$.

It is a known result that $\left(\sum_{j \neq l} \frac{p_j}{m}\right) + p_l$ is an upper bound on the makespan when using an arbitrary list scheduling algorithm if no precedence constraints need to be respected, where p_l is the job that makes the makespan, i.e., the job that finishes last. This result was shown in [30, 37] and is explained in detail in [60]. This upper bound can only be reached by a list scheduling rule if without job l each machine has the load $\sum_{j \neq l} \frac{p_j}{m}$, as only then job l is started at time $\sum_{j \neq l} \frac{p_j}{m}$ per definition of list scheduling.

The term $\left(\sum_{j \neq l} \frac{p_j}{m}\right) + p_l$ is maximal for the constructed instance if $n+1$ is the job that makes the makespan because $n+1$ is the longest job according to the assumption above. Thus, the following term is an upper bound on the maximum possible makespan for the constructed instance:

$$\left(\sum_{j \neq n+1} \frac{p_j}{m}\right) + p_{n+1} = \left(\sum_{j=1}^{n} \frac{p_j}{m}\right) + p_{n+1} = 2 \cdot \left(\sum_{j=1}^{n} \frac{p_j}{m}\right)$$

As already argued, this upper bound can be reached if and only if the jobs $1, \ldots, n$ can be scheduled such that each machine has a load of $\sum_{j=1}^{n} \frac{p_j}{m}$. This is possible if and only if the minimum makespan of the original problem is $\sum_{j=1}^{n} \frac{p_j}{m}$, which is a lower bound

on the makespan of the input instance. Thus, the makespan of $\overline{C}_{\max} = 2 \cdot \left(\sum_{j=1}^{n} \frac{p_j}{m}\right)$ for the constructed instance can be reached if and only if the minimum makespan of the $P||C_{\max}$ instance is $C^*_{\max} = \sum_{j=1}^{n} \frac{p_j}{m}$. This implies that $\overline{C}_{\max} > D$ holds if and only if the minimum makespan of the given $P||C_{\max}$ instance is $C^*_{\max} = \sum_{j=1}^{n} \frac{p_j}{m}$. Thus, the reduction is correct and complete. □

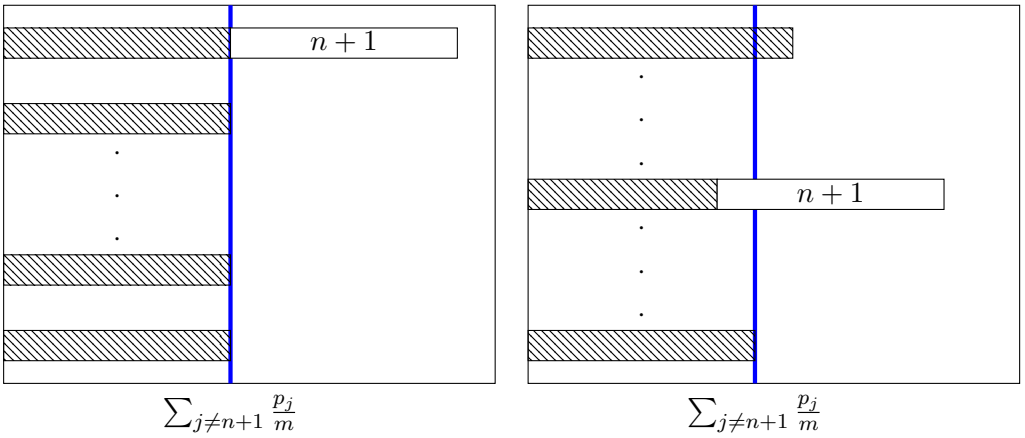


Figure 2.3 Example schedules constructed by list scheduling rules on the instance constructed in Theorem 6 if the minimum makespan of the original instance equals the average load (left) and if not (right).

The left part of Figure 2.3 illustrates the intuition behind the proof by exemplary showing that, if the original jobs can be equally distributed over the m machines, the new job can start at $\sum_{j \neq n+1} \frac{p_j}{m}$ and thus a makespan of $2 \cdot \left(\sum_{j \neq n+1} \frac{p_j}{m}\right)$ can be achieved by using a list scheduling rule. Additionally, the right side of the figure shows that, if the original jobs cannot be scheduled to achieve a makespan of the average load, there is a machine that finishes processing the original jobs before $\sum_{j \neq n+1} \frac{p_j}{m}$ and thus the list scheduling rule enforces job $n+1$ to start before that point in time as well. Therefore, a makespan smaller than $2 \cdot \left(\sum_{j \neq n+1} \frac{p_j}{m}\right)$ is achieved.

We will now use the strong NP-hardness of LS MAX to prove the strong NP-hardness of $\overline{\text{C-DAG}}$ by giving a polynomial time reduction from LS MAX to $\overline{\text{C-DAG}}$.

Theorem 7. $\overline{\text{C-DAG}}$ is strongly NP-hard.

Proof. Assume we are given an instance of LS MAX, i.e., $G = (V, E)$ with $V = \{1, \ldots, n\}$, processing times $p_j > 0$ for each $j \in V$, a number of machines m and a deadline D. We construct an instance of $\overline{\text{C-DAG}}$, i.e., a conditional DAG task $T = (G', p', \succ, m')$ and a deadline D', as follows:

1. Use the same number of machines as in the given instance (m).
2. For each node $j \in V$
   (a) Add $n = |V|$ copy nodes $v_j^1, \ldots, v_j^n$ of j each with a processing time of p_j.

(b) Add a conditional pair $(v_j, \overline{v_j})$ that uses each of the n job copies v_j^l as a conditional branch as illustrated in Figure 2.4.

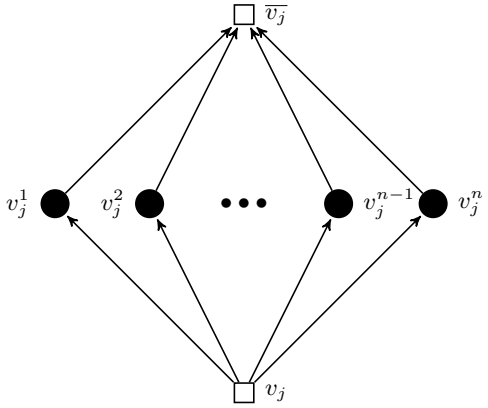


Figure 2.4 Example of a conditional pair as created in the proof of Theorem 7.

3. For each $(i, j) \in E$ introduce an edge from $\overline{v_i}$ to v_j.
4. Fix the priority order $\succ$ such that
   (a) $k < k'$ implies $v_j^k \succ v_i^{k'}$ for each $i, j \in V$.
   (b) $j \succ j'$ holds for all $j, j' \in V'$ with $p'_{j'} > 0 = p'_j$.
5. Use the same deadline as in the given instance (D).

Figure 2.5 illustrates the construction with a small example. In Step 2 of the reduction, we add a conditional pair as illustrated in Fig. 2.4 for each $j \in V$. Furthermore, in step 3 of the construction, two pairs are connected via an edge if the corresponding jobs in the original instance were connected by an edge.

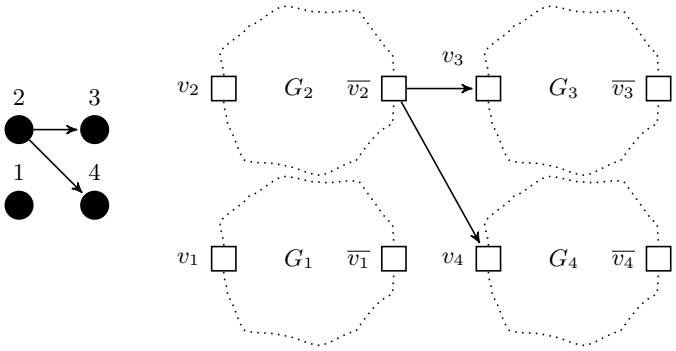


Figure 2.5 Example of the construction used to proof Theorem 7. Given an LS MAX instance with the precedence constraint graph as shown on the left side, the conditional DAG on the right side is constructed where each G_i is an abstraction of a conditional pair as illustrated in Figure 2.4.

We first argue that the construction can be executed in polynomial time and space. It introduces n^2 copy nodes and $2n$ conditional nodes. Additionally $2n^2$ edges are added to connect the copy and conditional nodes and $|E|$ edges are added to connect the conditional pairs. Therefore G' is of polynomial size.

To show correctness and completeness of the reduction, we show that $\overline{C}_{\max} = WCET(T')$ holds, where $\overline{C}_{\max}$ is the maximum makespan of the given LS MAX instance.

To do so, consider an arbitrary realization G'_r. Figure 2.6 illustrates a realization for the example in Fig. 2.5. Observe that, by definition, in G'_r exactly one job copy v_j^l is active for each job $j \in V$. Let v_j^l and $v_{j'}^{l'}$ be active job copies in G'_r, then by construction, there is a path $P = (v_j^l, \overline{v_j}, v_{j'}, v_{j'}^{l'})$ from v_j^l to $v_{j'}^{l'}$ in G'_r if and only if $(j, j') \in E$. Note that all vertices on P, apart from the endpoints, have a processing time of zero and that, according to the constructed order $\prec$, all vertices with processing time zero are processed before all other vertices. Therefore, the only function of P when scheduling G'_r is, that it formulates a precedence constraint between v_j^l and $v_{j'}^{l'}$. Thus, all precedence constraints between the original jobs in G are also present between the corresponding active job copies in G'_r.

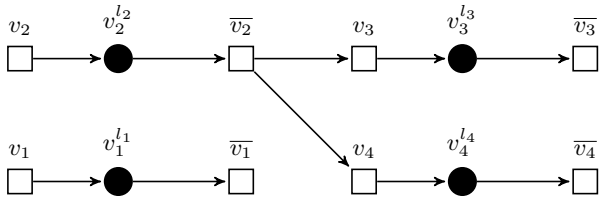


Figure 2.6 Example of a realization G'_r of the conditional DAG as shown in Figure 2.5 where $v_j^{l_j}$ denotes the active job copy of j in realization G'_r.

In addition to the active job copies and the paths that connect them, G'_r only contains a unique predecessor respectively successor with an execution time of zero for job copies v_j^l such that j has no predecessor respectively successor in G. As those jobs have an execution time of zero, they do not affect the schedule of G'_r given $\prec$. We can summarize that for G and each G'_r the following holds:

- There are precedence constraints between two active job copies v_j^l and $v_{j'}^{l'}$ in G'_r if and only if there are precedence constraints between j and j' in G.
- No additional precedence constraints and nodes affect the schedule of G'_r given $\prec$.
- G'_r and G are executed on the same number of machines.
- Each active job copy v_j^l has the same processing time as the original job j.

Thus, G and each G'_r effectively formulate the same scheduling instances that, apart from renaming, only can differ in the order the jobs are scheduled. Therefore, in order to show $\overline{C}_{\max} = WCET(T')$ it only remains to show the following two statements, where the first statement implies $WCET(T') \leq \overline{C}_{\max}$ and the second statement implies $\overline{C}_{\max} \leq WCET(T')$.

1. For each realization G'_r, there is a list scheduling order $\succ_{LS}$ that schedules the original jobs exactly as $\succ$ schedules the active job copies of G_r. That is, for each $j, j' \in V$ with $j \neq j'$ holds $j \succ_{LS} j'$ if and only if $v_j^l \succ v_{j'}^{l'}$ holds for the active job copies of j and j' in G'_r.

2. For each possible list scheduling order $\succ_{LS}$ on the original instance, there is a realization G'_r that schedules the active job copies the same way as $\succ_{LS}$ schedules the original jobs, i.e., for each pair of active job copies with $j \neq j'$ holds $v_j^l \succ v_{j'}^{l'}$ if and only if $j \succ_{LS} j'$ holds.

We start by showing the first statement. By definition of C-DAG, the active job copies in any realization G'_r are scheduled using fixed-priority scheduling with order $\prec$ to achieve a execution time of C_r. In LS MAX, any list scheduling order can be chosen. In specific, for each realization G'_r there is an order $\prec_{LS}$ that orders the jobs in G the same way as $\prec$ schedules the active job copies in G'_r. I.e., there is an order $\prec_{LS}$ such that $v_{j'}^{l'} \prec v_j^l$ holds for the active job copies v_j^l and $v_{j'}^{l'}$ of original jobs $j, j' \in V$ if and only if $j' \prec_{LS} j$. Thus, $\prec_{LS}$ achieves a makespan of C_r. As this holds for any realization, $WCET(T') \leq \overline{C}_{\max}$ follows.

To complete the proof, we show the second statement. Consider an arbitrary list scheduling order $\prec_{LS}$ on the jobs in G that achieves a makespan of C. We show that there is a realization G'_r such that $\prec$ orders the active job copies in G'_r exactly as $\prec_{LS}$ orders the original jobs in G. For each job $j \in V$ let q_j denote the position of j in $\prec_{LS}$, i.e., j has the q_j highest priority in V. Then, there is a realization G'_r such that $v_j^{q_j}$ is the sole active job copy of j in G'_r for each $j \in V$. Let j and j' with $j \neq j'$ be two arbitrary jobs in V with $j' \prec_{LS} j$, then $q_j < q_{j'}$ holds by definition and $v_j^{q_j}$ and $v_{j'}^{q_{j'}}$ are the active job copies in G'_r. By construction of $\prec$, $q_j < q_{j'}$ implies $v_{j'}^{q_{j'}} \prec v_j^{q_j}$. Thus, $\prec$ orders the active job copies in G'_r exactly as $\prec_{LS}$ orders the original jobs in G and achieves an execution time of C. As this holds for each list scheduling order $\prec_{LS}$, $\overline{C}_{\max} \leq WCET(T')$ and thus $\overline{C}_{\max} = WCET(T')$ follows. □

The previous theorem states that $\overline{\text{C-DAG}}$ is strongly NP-hard. Because, according to Theorem 1, $\overline{\text{C-DAG}} \in$ NP holds as well, the following theorem can be derived.

Theorem 8. $\overline{\text{C-DAG}}$ is strongly NP-complete.

Theorem 8, Lemma 1 and the definition of CoNP then imply the following theorem.

Theorem 9. C-DAG is strongly CoNP-complete.

2.3.2 Structure Preservation and Proof Framework

Observe that the introduced reduction from LS MAX to $\overline{\text{C-DAG}}$ in some way preserves the structure of the input precedence constraint graph G. Let $G' = (V', E', C')$ be the constructed conditional DAG and consider the graph $G_{C'}$ that uses each conditional pair $c_i = (v_i, \overline{v_i}) \in C'$ as a vertex and has edges between c_i and c_j if and only if $(\overline{v_i}, v_j) \in E'$. We can observe that G and $G_{C'}$ are isomorphic by definition of the reduction (see step 3). As each realization G'_r of G' just contains a simple path $(v_j, v_j^{l_j}, \overline{v_j})$ instead of each condition G_j and otherwise equals $G_{C'}$, we can observe the following lemma.

Lemma 3. Let $G = (V, E)$ be the precedence constraint graph of an LS MAX instance I and let $G' = (V', E', C')$ be the conditional DAG that is constructed by the reduction of Theorem 7 given I. Then, the following statements hold:

1. If G is a tree, then each realization G'_r of G' is a tree.
2. If G is a set of k disjoint chains, then each realization G'_r of G is a set of k disjoint chains.

This lemma and the reduction of Theorem 7 give us a proof framework to show the CoNP-hardness of $\overline{\text{C-DAG}}$ for special graph classes. If we show that LS MAX is NP-hard for precedence constraint graphs that form a tree or are a set of disjoint chains, Lemma 3 and the reduction of Theorem 7 imply the CoNP-hardness of the corresponding $\overline{\text{C-DAG}}$ variant.

We will exploit this proof framework in the Sections 2.4 and 2.5 to derive the corresponding hardness results.

2.3.3 Strong NP-Hardness: Preemption

Using Theorems 6 and 7 we were able to show that $\overline{\text{C-DAG}}$ is strongly NP-hard. Additionally, we can observe that the strong NP-hardness still holds if preemption is allowed. In contrast to the non-preemptive variant, we assume that schedule $S_T(G_r)$ of each realization G_r is defined to schedule all jobs using list scheduling with preemption based on $\succ$ for a given $T = (G, p, \succ, m)$. Recap, that list scheduling with preemption was defined in Section 1.3.2.

Theorem 10. $\overline{\text{C-DAG}}$ is strongly NP-complete even if preemption is allowed.

Proof. To show that $\overline{\text{C-DAG}}$ is still in NP if preemption is allowed, the same polynomial proof system as before can be used. It remains to show that $\overline{\text{C-DAG}}$ is still NP-hard.

Let f be the reduction used to prove Theorem 6 and let g be the reduction used to prove Theorem 7, i.e., f is the reduction from $P||C_{\max}$ to LS MAX and g is the reduction from LS MAX to $\overline{\text{C-DAG}}$. Then $g \circ f$ is a reduction that proves the strong NP-hardness of non-preemptive $\overline{\text{C-DAG}}$ by reducing from $P||C_{\max}$. This leaves to show that the reduction remains correct if preemption is allowed.

Per definition, f only creates LS MAX instances that do not have any precedence constraints. Let $T = (G, p, \succ, m)$ be the conditional DAG task constructed by $g \circ f$. Then, by definition of g, step 3. of the reduction does not introduce any edges. We can observe that all predecessors of jobs j with processing times greater than zero in a realization G_r have processing times of zero and thus, by construction of $\succ$, are all processed at point in time zero. It follows that each active job j with a processing time greater than zero becomes available at point in time zero in any realization G_r of G.

Thus, independent of the realization, no preemption occurs as no higher prioritized

job will ever become available while all machines are busy and lower prioritized jobs are being processed. Therefore, $g \circ f$ is still a valid reduction if preemption is allowed. □

Again, the previously shown theorem, Lemma 1 and the definition of CoNP imply the following theorem.

Theorem 11. C-DAG is strongly CoNP-complete even if preemption is allowed.

2.3.4 Weak NP-Hardness: Two Machines

The previous sections were able to prove the strong NP-hardness of $\overline{\text{C-DAG}}$ by reducing $P||C_{\max}$ to LS MAX and then LS MAX to $\overline{\text{C-DAG}}$.

We can observe, that in both reductions the number of machines in the constructed instances is the same as in the given instances. Because the number of machines remains the same, we can exploit the weak NP-hardness of $P2||C_{\max}$ in order to show that $\overline{\text{C-DAG}}$ is weakly NP-hard if the conditional DAG is executed on exactly two machines. $P2||C_{\max}$ is the variant of $P||C_{\max}$ where the number of machines is fixed at two and was shown to be weakly NP-hard in [39].

Corollary 1. $\overline{\text{C-DAG}}$ is weakly NP-complete even if the number of machines is fixed at two.

Proof. The statement can be shown by first reducing $P2||C_{\max}$ to LS MAX on two machines using the same reduction as in the proof of Theorem 6.

Then, we can reduce LS MAX on two machines to $\overline{\text{C-DAG}}$ on two machines by using the same reduction as in the proof of Theorem 7. □

Again, the previously shown corollary, Lemma 1 and the definition of CoNP imply the following corollary.

Corollary 2. C-DAG is weakly CoNP-complete even if the number of machines is fixed at two.

2.3.5 Strong NP-Hardness: Series-Parallel Graphs

In this section, we will adjust the reduction of Theorem 7 to show that $\overline{\text{C-DAG}}$ is strongly NP-hard even if the given conditional DAGs are two-terminals series-parallel graphs. To do so, we expand the reduction by the following steps:

6. Add a source s and a sink t with processing times of zero to the conditional DAG.
7. For each j such that no $j' \in V$ with $(j', j) \in E$ exists, add an edge from s to v_j.
8. For each j such that no $j' \in V$ with $(j, j') \in E$ exists, add an edge from $\overline{v_j}$ to t.

We can observe that this steps do not affect the completeness and correctness of the reduction as adding dummy terminals with processing times of zero does not affect the

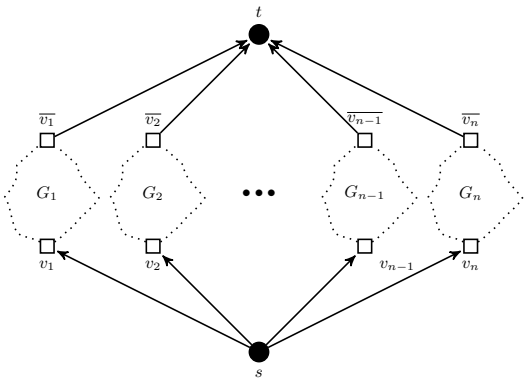


Figure 2.7 Graph obtained by the reduction in the proof of Theorem 12 where G_j illustrates the conditional branches of the conditional pair c_j.

execution time of any realization and thus the worst-case execution time.

In order to show that the NP-hardness result for $\overline{\text{C-DAG}}$ holds even if only conditional DAGs that are two-terminals series-parallel graphs are considered, the following alternative and equivalent definition of series-parallel graphs is used as. The alternative definition was introduced and the equivalence shown in [59].

Lemma 4. A directed two-terminals graph $G = (V, E)$ is *series-parallel* if and only if it can be reduced to a single edge using an appropriate sequence of the following reduction rules:

1. Replace two edges (u, v) and (v, w), where v has a degree of two, with a single edge (u, w).
2. Replace two parallel edges (u, v) and (u, v) with a single edge with the same endpoints.

Theorem 12. $\overline{\text{C-DAG}}$ is strongly NP-complete even if G is a two-terminals series-parallel graph.

Proof. Note that the problem is still in NP because the same polynomial proof system as before can be used. Thus, it remains to show the strong NP-hardness. Let f be the reduction used to prove Theorem 6 and let g be the reduction used to prove Theorem 7, with the additional steps as defined earlier in this section. Then $g \circ f$ is a reduction that proves the NP-hardness of $\overline{\text{C-DAG}}$.

To show the theorem, it is sufficient to show that each conditional DAG that is constructed by $g \circ f$ is a two-terminals series-parallel graph. Let G be an arbitrary graph that is constructed using $g \circ f$. In the following, a sequence of reduction rules will be given that reduces G to a single edge and thus, according to lemma 4, proves that G is a series-parallel graph.

Because f only constructs LS MAX instances without any precedence constraints, step

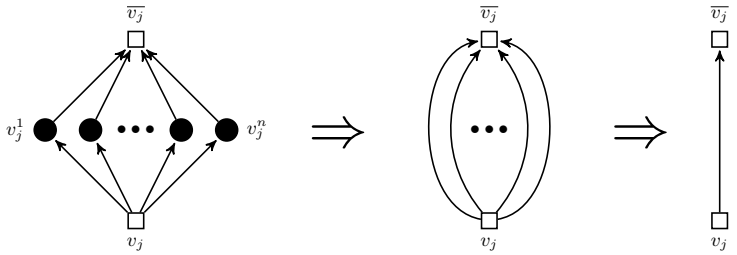


Figure 2.8 Illustration of how each conditional pair $(v_j, \overline{v_j})$ that is constructed by the reduction of Theorem 7 can be reduced to a single edge using the reduction rules as defined in Lemma 4.

3 of g does not introduce any edges and thus G has the form as illustrated in Figure 2.7.

Per construction, each conditional branch of an arbitrary conditional pair $(v_j, \overline{v_j})$, is just a single node v_j^l with a degree of two and incident edges (v_j, v_j^l) respectively $(v_j^l, \overline{v_j})$. Using the first reduction rule of Lemma 4, each of those branches can be replaced by a single edge $(v_j, \overline{v_j})$. Afterwards, all the parallel edges $(v_j, \overline{v_j})$ can be replaced by a single edge $(v_j, \overline{v_j})$. Figure 2.8 illustrates the described sequence of reduction rules. As each conditional branch can be reduced to a single edge, G can be reduced to the form as illustrated in Figure 2.9. Then, each s-t-path in that graph can be reduced to a single edge (s, t) by again using the first reduction rule. Finally, all created parallel edges (s, t) can be replaced by a single edge (s, t) using the second reduction rule. Thus, G can be reduced to a single edge and thus is a two-terminals series-parallel graph.

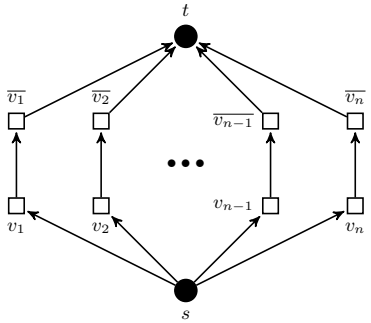


Figure 2.9 Graph that can be obtained using reduction rules from the graph in Figure 2.7.

□

Again, the previously shown theorem, Lemma 1 and the definition of CoNP imply the following theorem.

Theorem 13. C-DAG is strongly CoNP-complete even if G is a two-terminals series-parallel graph.

2.4 Chain Realizations

In this section, we prove the NP-hardness of $\overline{\text{C-DAG}}$ for a more restricted class of conditional DAGs. In specific, this section will consider conditional DAGs G for which holds that every realization G_r is a constant number of disjoint chains.

Before we discuss hardness results, consider the example of Figure 2.10 that again shows how source code can be modeled using conditional DAGs. Observe, that each realization of both components is a single chain and thus each realization of the complete conditional DAG is a set of disjoint chains. We remark that this way of modeling source code with conditional DAGs corresponds to a special case of control flow graphs as introduced by Allen [2]. Therefore, as control flow graphs are a common representation of source code, results regarding conditional DAGs with chain realizations might be of practical relevance.

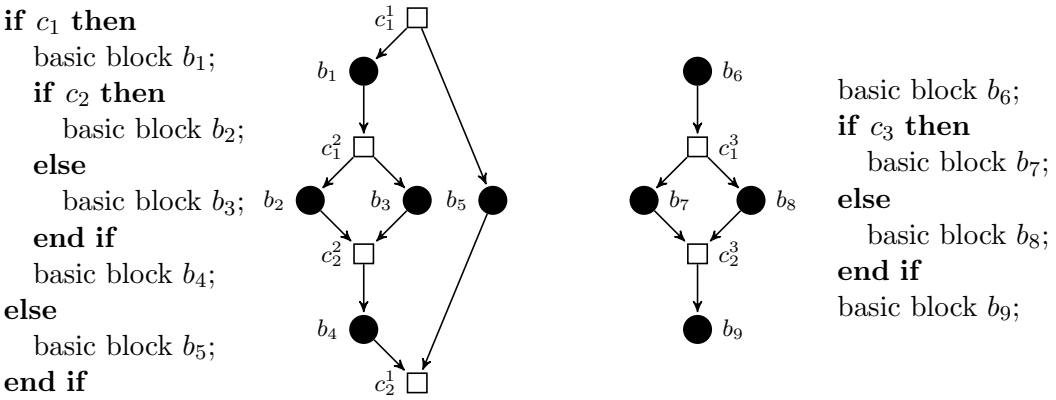


Figure 2.10 Two example source code excerpts and their representation as conditional DAGs. Each basic block b_i represents a sequence of statements that is guaranteed to be executed sequentially and each c_i represents some expression.

To derive hardness results for this special case, the proof framework as presented in Section 2.3.2 will be used. Therefore, the first subsection will consider the NP-hardness of LS MAX for precedence constraint graphs that are a constant number of chains. Finally, the second subsection will use that result to show the NP-hardness of $\overline{\text{C-DAG}}$ for the considered class of conditional DAGs.

2.4.1 List Scheduling Maximization

In the previous NP-hardness proofs for LS MAX, reductions from $P||C_{\max}$ were used. For the case of precedence constraint graphs that are chains, we again use a reduction from a variant of $P|prec|C_{\max}$.

Agnetis, Flamini, Nicosia, and Pacifici [1] showed that $P|prec|C_{\max}$ is weakly NP-hard even if the precedence constraints are three disjoint chains and two machines are used to process the jobs. We will use that result in order to show the NP-hardness of a

corresponding LS MAX variant that only considers precedence constraint graphs which are a constant number of disjoint chains. At that, we want to prove the NP-hardness of the smallest possible case.

We first can observe that the problem is trivial if $m \geq k$ holds, where m is the number of machines and k is the constant number of disjoint chains. This is the case as, independent of the used list scheduling order, all jobs would just be scheduled directly after they become available. Let $c_1, \ldots, c_k$ be the chains of such an instance and let $p(c_i)$ be the sum the of processing times of all jobs on chain c_i, then $\overline{C}_{\max} = \max_{i \in \{1,\ldots,k\}} p(c_i)$ always holds for $m \geq k$. Additionally, the problem is trivial on a single machine because then the maximum makespan is just the sum of all processing times

Therefore, we only need to consider cases with $1 < m < k$. Note that, as k is a constant, the number of machines is bounded by a constant in all of the considered cases. Thus, the smallest remaining non-trivial case is $m = 2$ and $k = 3$. However, LS MAX is solvable in polynomial time for three chains on two machines.

To show that this is indeed the case, we first prove the following lemma. The lemma intuitively states that idle time in list schedules of instances with precedence constraints that are a set of disjoint chains can only occur at points in time t at which the number of chains that still need processing is smaller than the number of machines.

Lemma 5. Let $J = \{1, \ldots, n\}$ be a set of jobs, p_j the processing time for each $j \in J$ and $G = (V, E)$ with $V = J$ a precedence constraint graph such that G has the form of k disjoint chains. Let $m < k$ be the number of identical machines. A list schedule of the instance has idle time at point in time t if and only if all jobs of at least $(k - m) + 1$ chains are completed at t.

Proof. To show the first direction, assume we are given an instance as described in the lemma and assume there was a list scheduling schedule with idle time before all jobs of at least $(k - m) + 1$ chains are completed. Then, there is a point in time t at which the idle time starts on a machine l and per assumption at least m chains have not been completely processed yet. Let $j_1, \ldots, j_m$ be the first jobs in those chains that have not been completely processed yet.

Per definition of list scheduling, the only reason that none of the jobs $j_1, \ldots, j_m$ is scheduled to start at time t on machine l is, that all of those jobs are currently being processed on other machines. As we have m jobs but, because l is idle at time t, only at most $m - 1$ machines that are busy at time t, it is not possible that all of the jobs are being processed at time t. Thus, the idle time cannot occur which contradicts the assumption. Therefore, if there is idle time at point in time t, at least $(k-m)+1$ chains are completed at t.

To show the second direction, assume that at least $(k - m) + 1$ chains are completely processed at point in time t in a list schedule. Then, at t only $m' < m$ chains still need

processing. As only one job per chain can be processed at the same point in time, only m' jobs can be processed at t. Thus, there is idle time at t. □

Theorem 14. LS MAX is solvable in polynomial time for three disjoint chains on two machines.

Proof. Let c_1, c_2 and c_3 be the given disjoint chains and let $p(c_i)$ denote the cumulative processing times of all jobs on chain c_i. Assume without loss of generality that $p(c_1) \leq p(c_2) \leq p(c_3)$ holds.

We now claim that no list scheduling rule with a makespan greater than $p(c_1) + p(c_3)$ can exist. Note that, if one machine has a load of $p(c_1) + p(c_3)$, the other machine has a load of $p(c_2)$. In order to prove the claim, assume there was a list scheduling rule such that one machine, say m_1, has a load of strictly more than $p(c_1) + p(c_3)$. Then, machine m_2 must have a load of strictly less than $p(c_2)$. Thus m_2 is only busy until some point in time $t < p(c_2)$ and idle afterwards. Observe that no c_i can finish earlier than at point in time $p(c_i)$, because c_i is a chain. As $t < p(c_2) \leq p(c_3)$ holds, only c_1 can be finished at time t. Therefore the idle time on machine m_2 after t contradicts Lemma 5 and thus no schedule with a makespan greater than $p(c_1) + p(c_3)$ can exist.

To finish the proof, we observe that by using the list scheduling rule that first schedules all jobs of c_1, then all jobs of c_2 and finally all jobs of c_3, a makespan of $p(c_1) + p(c_3)$ is achieved as illustrated by Figure 2.11. Thus, the maximum makespan is always $p(c_1) + p(c_3)$ and can therefore be determined in polynomial time.

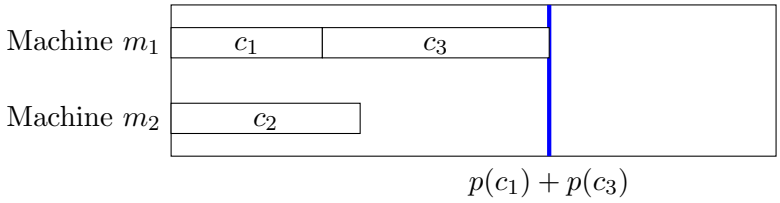


Figure 2.11 Illustration used in the proof of Theorem 14.

□

Because of Theorem 14 we know that, unless P = NP, we cannot hope to show the hardness of LS MAX for three or less chains on two machines. Thus, the smallest hardness result we can hope to obtain is for four chains on two machines.

Theorem 15. LS MAX is weakly NP-hard for four disjoint chains on two machines.

Proof. We will proof the NP-hardness by reduction from $P|prec|C_{\max}$ for three disjoint chains on two machines. Let therefore $J = \{1, \ldots, n\}$ be the set of jobs, $p_j > 0$ for each $j \in J$ the processing times and c_1, c_2 and c_3 the three disjoint chains that together contain all jobs.

We assume without loss of generality that for each chain the cumulative processing

time of all jobs in the chain is strictly less than $\frac{1}{2} \cdot (\sum_{j \in J} p_j)$. By doing so, we just exclude the trivial case where the minimum makespan is the size of the longest chain.

We construct an instance of LS MAX for four chains on two machines by using the same jobs and chains as in the given instance but adding a job $n+1$ with $p_{n+1} = \frac{1}{2} \cdot (\sum_{j \in J} p_j)$ that forms a single new chain c_4. Finally, we use $D = (\sum_{j \in J} p_j) - 1$ as deadline. The resulting instance has four chains as it does add exactly one additional chain and can clearly be constructed in polynomial time.

To prove the correctness and completeness of the reduction, we show that the maximum makespan in the constructed instance is $\overline{C}_{\max} = \sum_{j \in J} p_j$ if and only if the minimum makespan of the input instance is $C^*_{\max} = \frac{1}{2} \cdot (\sum_{j \in J} p_j)$. In the following, we proof both directions separately.

Assume the minimum makespan of the input instance is $\frac{1}{2} \cdot (\sum_{j \in J} p_j)$, then we can construct a list scheduling rule that just uses the order that was used to achieve the minimum makespan and schedules the new job $n+1$ last. The resulting schedule then reaches a makespan of $\sum_{j \in J} p_j$.

It remains to show that there is no list scheduling rule that leads to a larger makespan. Therefore, assume that there is a list scheduling rule that achieves a makespan of $C > \sum_{j \in J} p_j$ and let m_1 be the machine with the higher load in that schedule. As m_1 has a higher load than $\sum_{j \in J} p_j$, it follows that $n+1$ must be scheduled on m_1 in addition to original jobs with cumulative processing times of more than $\frac{1}{2} \cdot (\sum_{j \in J} p_j)$. Consequently m_2 has a load of less than $\frac{1}{2} \cdot (\sum_{j \in J} p_j)$.

We now argue that $n+1$ must be the last job scheduled on m_1. Therefore assume $n+1$ was not the last job on m_1, then at least a job j exists that is scheduled after $n+1$ and per construction is part of a different chain than $n+1$. Because per Lemma 5 there is no idle time until all but one chains are finished, j can only be scheduled after $n+1$ if m_2 is busy the entire time. But, as m_2 only has a load of less than $\frac{1}{2} \cdot (\sum_{j \in J} p_j)$ and $n+1$ alone has a processing time of $\frac{1}{2} \cdot (\sum_{j \in J} p_j)$, this is not the case and therefore j cannot be scheduled after $n+1$. Thus $n+1$ must be the last job on m_1. Figure 2.12 visualizes this argument.

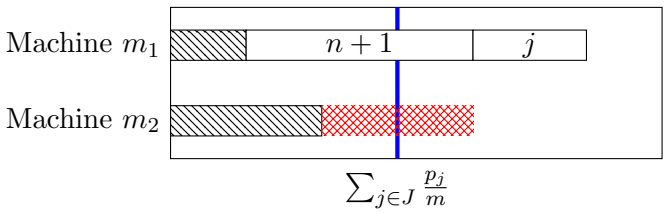


Figure 2.12 Illustration used in the proof of Theorem 15. The red colored area marks idle time that cannot occur according to Lemma 5.

But if $n+1$ is the last job on machine m_1 and starts at a point in time later than $\frac{1}{2} \cdot (\sum_{j \in J} p_j)$ while m_2 is only busy up to a point in time earlier than $\frac{1}{2} \cdot (\sum_{j \in J} p_j)$,

there is idle time on machine m_2 before all but one chains are finished as illustrated in Figure 2.13. This contradicts Lemma 5 and thus a schedule with a makespan larger than $\sum_{j \in J} p_j$ cannot exist.

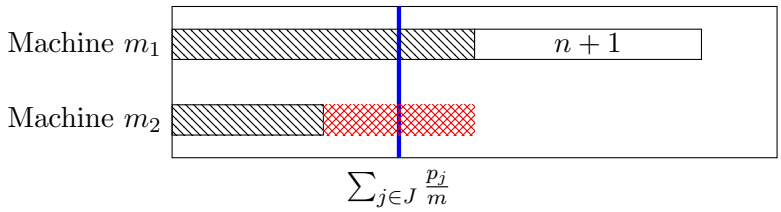


Figure 2.13 Illustration used in the proof of Theorem 15. The red colored area marks idle time that cannot occur according to Lemma 5.

To prove the other direction, assume that the maximum makespan of the constructed instance is $\sum_{j \in J} p_j$. This can only happen if all original jobs are scheduled on the same machine or job $n+1$ is scheduled on a machine together with additional jobs with cumulative processing times of exactly $\frac{1}{2} \cdot (\sum_{j \in J} p_j)$.

If all original jobs are scheduled on the same machine, $n+1$ must be scheduled on the other machine, say m_2, at time zero. As per Lemma 5 there is only idle time if all but one chains are completely processed, all jobs of two of the original chains must have been completely processed when $n+1$ is finished, i.e., at time $\frac{1}{2} \cdot (\sum_{j \in J} p_j)$. In order to reach the makespan of $\sum_{j \in J} p_j$ the remaining unfinished chain must have a cumulative processing time of at least $\frac{1}{2} \cdot (\sum_{j \in J} p_j)$, which contradicts the assumption from the very beginning of the proof. Thus it cannot happen that every original job is scheduled on the same machine.

If job $n+1$ is scheduled on a machine, say m_1, together with additional jobs with a cumulative makespan of exactly $\frac{1}{2} \cdot (\sum_{j \in J} p_j)$, there must be original jobs from at least two chains on the machine per assumption about the chain size. Because this is the case, $n+1$ must be the last job on m_1 as otherwise idle time would occur before all but one chains are completely processed, which contradicts Lemma 5. This means that $n+1$ starts at $\frac{1}{2} \cdot (\sum_{j \in J} p_j)$ and thus the minimum makspan of the input problem is $\frac{1}{2} \cdot (\sum_{j \in J} p_j)$ as illustrated in Figure 2.14. Thus, $\overline{C}_{\max} = \sum_{j \in J} p_j$ holds if and only if the original three chains can be scheduled to achieve a minimum makespan of $C^*_{\max} = \frac{1}{2} \cdot (\sum_{j \in J} p_j)$.

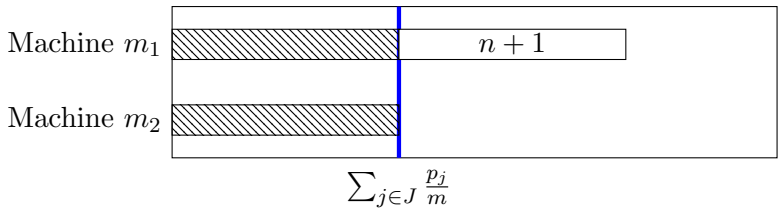


Figure 2.14 Illustration used in the proof of Theorem 15.

It follows that $\overline{C}_{\max} > (\sum_{j \in J} p_j) - 1 = D$ holds if and only if the original three chains can be distributed equally on the machines, which proves the correctness and completeness of the reduction. □

2.4.2 Weak NP-Hardness

We now use the results of the previous subsection to show the CoNP-hardness of C-DAG for conditional DAGs whose realizations are a constant number of disjoint chains. As usual, we will do so by showing that $\overline{\text{C-DAG}}$ is NP-hard even if all realizations are a constant number of disjoint chains.

Theorem 16. $\overline{\text{C-DAG}}$ is weakly NP-complete even if each realization has the form of four disjoint chains and the number of machines is fixed at two.

Proof. To show that this $\overline{\text{C-DAG}}$ variant is still in NP, the same polynomial proof system as before can be used. It remains to show the NP-hardness.

Let g be the reduction used in the prove of Theorem 7, i.e., the reduction from LS MAX to $\overline{\text{C-DAG}}$, and let f be the reduction used to prove Theorem 15. Then, per transitivity, $g \circ f$ is a polynomial time reduction from the load balancing problem for three chains on two machines to $\overline{\text{C-DAG}}$.

To prove the theorem it only remains to show that instances constructed by $g \circ f$ have the required structure. As g only accepts instances on two machines and neither f nor g change the number of machines, the constructed instance only uses two machines. Because f only creates instances such that their precedence constraint graphs consist of four chains, each input G of g in the reduction $g \circ f$ has the form of four disjoint chains. Per Lemma 3 we know that each realization G_r then consists of four disjoint chains as well. Thus, $g \circ f$ is a polynomial time reduction from the load balancing problem for three chains on two machines to the $\overline{\text{C-DAG}}$ variant and the theorem follows. □

As usual, the weak NP-hardness of the $\overline{\text{C-DAG}}$ variant, Lemma 1 and the definition of CoNP imply the following theorem.

Theorem 17. C-DAG is weakly CoNP-complete even if each realization has the form of four disjoint chains and the number of machines is fixed at two.

So far, we have seen that the worst-case execution time problem for conditional DAGs is NP- respectively CoNP-hard even if each realization has the form of four chains and two machines are used to process the jobs. Section 3.1.1 will show that the problem is solvable in polynomial time on a single machine. Additionally, for similar reasons as argued for LS MAX, we can observe that the problem is also solvable in polynomial time if $k \leq m$ holds. This is the case as the worst-case execution time then just is the cumulative processing time of the longest path in the conditional DAG, which can be computed in polynomial time, as shown for example in [55].

Therefore, the smallest relevant case is the problem for conditional DAGs such that each realization has the form of three chains. As LS MAX for three chains is solvable in polynomial time, we unfortunately cannot hope to proof the hardness of $\overline{\text{C-DAG}}$ for that class of conditional DAGs by using the same proof framework as before.

Finally, we can observe that if the number of chains is not constant, the problem is indeed strongly CoNP- respectively NP-hard because the composition of the reductions used in the Theorems 6 and 7 constructs conditional DAGs whose realizations are n chains.

2.5 Tree Realizations: NP-Hardness

Another structure that is often considered in the context of scheduling, is the structure of trees. Therefore, the aim of this section is to investigate whether the restriction that every realization of the given conditional DAG must be a tree changes the previously obtained complexity results. In specific, we will see that the problem is still strongly NP- respectively CoNP-hard even if we add said restriction.

As in the previous section, we derive these results by using the proof framework of Section 2.3.2. Therefore, we first show that LS MAX is NP-hard even if the precedence constraints form a tree. Afterwards, Lemma 3 and the reduction of Theorem 7 will again be exploited in order to show that $\overline{\text{C-DAG}}$ and C-DAG are still NP- respectively CoNP-hard even if every realization is a tree.

Theorem 18. LS MAX is strongly NP-hard even if the precedence constraint graph is a tree.

Proof. We show the theorem by reducing from $P||C_{\max}$. Let therefore $J = \{1, \dots, n\}$ be the set of jobs, let $p_j > 0$ be the processing time for $j \in J$ and let m be the number of identical machines.

We construct an instance of the list scheduling makespan maximization problem for trees by using the same number of machines. Then, we use the same set of jobs with the same processing times as in the given instance, but add a job $n+1$ with a processing time of $p_{n+1} = \sum_{j=1}^{n} \frac{p_j}{m}$. To obtain an instance such that the precedence constraint graph is a tree, we additionally add a root job r with a processing time of one and outgoing edges to all existing jobs. In summary, we construct the instance $G = (V, E)$ with $V = J \cup \{n+1, r\}$ and $E = \{(r, v) \mid v \in (V \setminus \{r\})\}$ with processing times p_j as in the given instance for each $j \in J$, $p_r = 1$ and $p_{n+1} = \sum_{j=1}^{n} \frac{p_j}{m}$. Additionally, we use $D = 2 \cdot (\sum_{j=1}^{n} \frac{p_j}{m})$ as deadline. As Figure 2.15 illustrates, the constructed instance is a tree – in specific, an out-tree – because it has the single root r and each other node is reachable from r by exactly one path.

In order to execute the construction, two additional jobs and $n+1$ edges must be

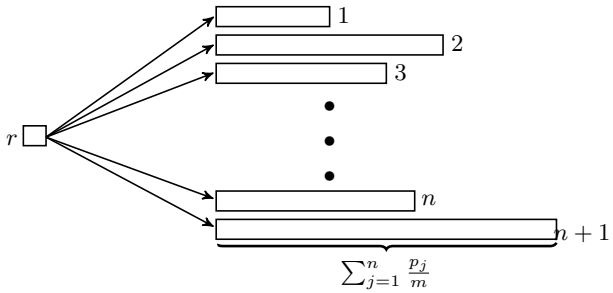


Figure 2.15 Illustration of an LS MAX instance for trees as constructed in the proof of Theorem 18.

added. Additionally, $p_{n+1} = \sum_{j=1}^{n} \frac{p_j}{m}$ needs to be calculated. As all of this steps can be executed in polynomial time, the reduction can be executed in polynomial time as well. To show the correctness and completeness of the reduction, we show that the maximum makespan of the constructed instance is $\overline{C}_{\max} = 1 + 2 \cdot (\sum_{j=1}^{n} \frac{p_j}{m})$ if and only if the minimum makespan of the original instance is $C^*_{\max} = \sum_{j=1}^{n} \frac{p_j}{m}$. We again assume without loss of generality that $p_j \leq \sum_{j=1}^{n} \frac{p_j}{m}$ holds for each $j \in J$.

To prove the statement, observe that each list scheduling schedule must start by executing r as every other job only becomes available after r is finished. Additionally, for the same reason, it is not possible that any other job is processed in parallel to r.

As every other job is available directly after r, it follows that the maximum makespan of the constructed instance is $\overline{C}_{\max} = 1 + \overline{C'}_{\max}$ where $\overline{C'}_{\max}$ is the maximum makespan that can be obtained by a list scheduling rule on the instance $J \cup \{n+1\}$ with the introduced processing times but without precedence constraints.

Now, in the proof of Theorem 6 we already showed that the maximum makespan of the instance $J \cup \{n+1\}$ is $\overline{C'}_{\max} = 2 \cdot (\sum_{j=1}^{n} \frac{p_j}{m})$ if and only if the minimum makespan of instance J is $C^*_{\max} = \sum_{j=1}^{n} \frac{p_j}{m}$.

As the maximum makespan of the constructed instance is $\overline{C}_{\max} = 1 + \overline{C'}_{\max}$ it follows that $\overline{C}_{\max} = 1 + 2 \cdot (\sum_{j=1}^{n} \frac{p_j}{m})$ holds if and only if $\overline{C'}_{\max} = 2 \cdot (\sum_{j=1}^{n} \frac{p_j}{m})$ holds. This holds if and only if the minimum makespan of the input problem is $C^*_{\max} = \sum_{j=1}^{n} \frac{p_j}{m}$. Figure 2.16 illustrates a list scheduling schedule of the constructed instance with a makespan of $1 + 2 \cdot (\sum_{j=1}^{n} \frac{p_j}{m})$.

It follows that $\overline{C}_{\max} > D$ holds if and only if the minimum makespan of the input problem is $C^*_{\max} = \sum_{j=1}^{n} \frac{p_j}{m}$, which proves correctness and completeness of the reduction. □

We now show the CoNP-hardness of C-DAG for conditional DAGs G such that each realization G_r of G is a tree. As usual, we do so by considering the complement of the problem. Again, we use the previously shown result in combination with Theorem 7 and Lemma 3 to show the following theorem.

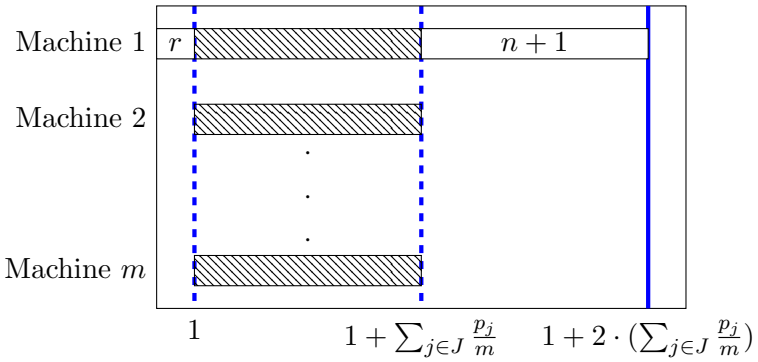


Figure 2.16 Illustration of an list scheduling schedule for the constructed instance of Theorem 18 with a makespan of $1+2\cdot(\sum_{j\in J}\frac{p_j}{m})$.

Theorem 19. $\overline{\text{C-DAG}}$ is strongly NP-complete even if each realization of the given conditional DAG is a tree.

Proof. To show that this $\overline{\text{C-DAG}}$ variant is still in NP, the same polynomial proof system as before can be used. It remains to show the NP-hardness.

Let f be the reduction of Theorem 18 and let g be the reduction of Theorem 7, then per transitivity, $g \circ f$ is a polynomial time reduction from a $P||C_{\max}$ to $\overline{\text{C-DAG}}$. It therefore only remains to show that each realization of any conditional DAG constructed by $g \circ f$ is a tree. Per proof of Theorem 18, f only creates instances of LS MAX such that the precedence constraints form a tree. Per Lemma 3 each realization of a conditional DAG constructed by g then is a tree as well. □

Again, Theorem 19, Lemma 1 and the definition of CoNP imply the following theorem.

Theorem 20. C-DAG is strongly CoNP-complete even if each realization is a tree.

To conclude this section, consider a variant of the problem where the number of machines is a constant. Per Section 3.1.1 the problem is solvable in polynomial time if the constant number of machines is one. Thus, the smallest case of interest is the case of two identical parallel machines. Before considering this case, observe that the reduction of Theorem 7 and 18 neither depend on nor change the number of machines of the given instance. According to [39], the variant of $P||C_{\max}$ on two identical machines ($P2||C_{\max}$) is weakly NP-hard as well. Thus, the whole argumentation of this section can be made using $P2||C_{\max}$ instead of $P||C_{\max}$, which leads to the following corollary.

Corollary 3. $\overline{\text{C-DAG}}$ and C-DAG are weakly NP- respectively CoNP-complete even if each realization is a tree and the number of machines is fixed at two.

Chapter 3

Algorithms

In Chapter 2 the complexity of the worst-case execution time problem for conditional DAGs given a fixed-priority order was discussed. In specific, the two decision problems $\overline{\text{C-DAG}}$ and C-DAG were considered and shown to be strongly NP- respectively CoNP-hard in the general case and for more restricted classes of conditional DAGs. As this is the case, we cannot hope to find exact polynomial, or even pseudo-polynomial, time algorithms for the general case of both problems, unless P = NP.

Nevertheless, polynomial and pseudo-polynomial time algorithms are still possible for some special cases of the problem. This cases will be discussed in Section 3.1 and algorithms will be derived accordingly.

As exact polynomial and pseudo-polynomial time algorithms are only possible for special cases, Section 3.2 considers approximations for more general variants of the problem. Therefore, we define optimization problem variants for the two decision problems and derive approximations. In the process, a fully polynomial-time approximation scheme (FPTAS) for a special case of both approximation problems will be presented. The FPTAS exploits the pseudo-polynomial time algorithm as introduced in Section 3.1.2 in a rounding approach.

Finally, the chapter will conclude by considering another variant of C-DAG that only allows fixed-priority orders which fulfill some additional constraints. Section 3.3 defines these constraints and shows that the worst-case execution time of this variant can be computed in polynomial time for some cases that are CoNP-hard for arbitrary fixed-priority orders.

3.1 Exact Algorithms

We start by discussing exact algorithms that compute the worst-case execution time for conditional DAGs given a fixed-priority order. As $\overline{\text{C-DAG}}$ and C-DAG were shown to be NP- respectively CoNP-hard in the general case and for more restricted classes of conditional DAGs, we cannot hope to find polynomial time algorithms for those problems, unless P = NP. Nevertheless, the hardness of both problems was shown for the

case where the number of machines is either part of the input, see Theorems 7 and 9, or constant at value two, see Corollaries 1 and 2. Therefore, an exact polynomial time algorithm that computes the worst-case execution time for a given conditional DAG and priority order on a single machines is still possible and given in Section 3.1.1. The discussed algorithm was first introduced in [46]. We will see that this algorithm relies on the independence of different (non-nested) conditions and thus does not contradict Theorem 5, i.e., the strong CoNP-hardness of C-DAG for conditional DAGs with shared nodes on a single machine.

Furthermore, Chapter 2 showed that $\overline{\text{C-DAG}}$ and C-DAG are even strongly NP- respectively CoNP-hard in the general case and for some of the more restricted classes of conditional DAGs. Thus, not even pseudo-polynomial time algorithms that compute the worst-case execution time for those classes of conditional DAGs are possible, again unless P = NP. On the contrary, both decision problems were only shown to be weakly NP- respectively CoNP-hard if each realization of the given conditional DAG is a constant number of disjoint chains. Therefore, pseudo-polynomial time algorithms are possible for this cases and will be considered in Section 3.1.2. To be more precise, we introduce a pseudo-polynomial time dynamic program for conditional DAGs G such that each realization of G has a bounded width.

3.1.1 Single Machine Worst-Case Execution Time

This section discusses the worst-case execution time problem for a conditional DAG task on a single machine, where a conditional DAG task $T = (G, p, \succ, 1)$ is given and the goal is to calculate $WCET(T) = \max_{r \in \mathcal{R}_G} C_T(G_r)$.

Therefore, we first can observe that the execution time of a realization G_r on a single machine is just the sum of processing times over all active jobs in G_r. This is the case as all active jobs need to be processed on a single machine and no idle time can occur by definition of schedule $S_T(G_r)$. Therefore, the following equality holds for a conditional DAG task T with $m = 1$:

$$WCET(T) = \max_{r \in \mathcal{R}_G} C_T(G_r) = \max_{r \in \mathcal{R}_G} \sum_{j \in V_r} p_j$$

We can conclude that the execution time does not depend on the priority order of the jobs but only on the activity of the conditional branches. Therefore, we ignore the priority order for the remainder of this section.

In the following, we will see that the term $\max_{r \in \mathcal{R}_G} \sum_{j \in V_r} p_j$ for a conditional DAG task T, and thus the worst-case execution time on a single machine, can be computed in polynomial time. Therefore, the polynomial time algorithm for this problem as introduced in [46] will be considered. At that, we will assume that the given conditional DAG G has a single source s and sink t. This can be done without loss of generality

because adding a dummy source and sink with processing times of zero to a conditional DAG does not affect the execution times of the realizations.

The algorithm first computes a *topological order* of the given conditional DAG as defined in Definition 35. The topological order of a DAG can be calculated in $\mathcal{O}(|V|+|E|)$ by for example using *Kahn's Algorithm* [35].

Definition 35. Given a directed acyclic graph $G = (V, E)$ with $|V| = n$, an ordering $v_1, v_2, \ldots, v_n$ of the vertices is a *topological order* if $i < j$ holds for each $(v_i, v_j) \in E$.

Theorem 21. The worst-case execution time of a conditional DAG task on a single machine can be computed in polynomial time.

Proof. We will prove the theorem by giving the algorithm that calculates $WCET(T) = \max_{r \in \mathcal{R}_G} \left(\sum_{j \in V_r} p_j \right)$ for a given conditional DAG task $T = (G, p, \succ, 1)$ as introduced in [46].

Assume that the vertices of the conditional DAG are indexed according to the topological order. The algorithm will now calculate the dynamic programming table D where $D[i]$ for all $i \in \{1, \ldots, n\}$ denotes the set of active jobs that maximizes the worst-case execution time for processing the subgraph that is induced by v_i and all, direct and indirect, successors of v_i in G. Let V_i be the set of v_i and all, direct and indirect, successors of v_i, then $D[i]$ contains $A_i \subseteq V_i$ with

$$A_i = \text{argmax}_{V_i' \in \{V' \in \mathcal{P}(V_i) | \exists r \in \mathcal{R}_G : V' \subseteq V_r\}} \sum_{j \in V_i'} p_j$$

where $\mathcal{P}(V_i)$ denotes the power set of V_i.

Starting with $D[n]$ where n is the index of the last element of the topological order, i.e., of the sink t, one can use equation (3.1) to iteratively calculate the values for all dynamic programming cells, where $P(D[j])$ denotes the cumulative processing times over the jobs in $D[j]$ and $succ(v_j)$ denotes the set of direct successors of v_j. As $P(D[j])$ remains constant once $D[j]$ is calculated, we define the algorithm to compute and cache it at that point such that it has to be determined exactly one time. In the end, $D[1]$ contains the set of active nodes that maximizes the makespan for the complete conditional DAG task. Thus, the algorithm can return $WCET(T) = \sum_{j \in D[1]} p_j$.

$$D[i] = \begin{cases} \{v_i\} \cup D[\text{argmax}_{j : v_j \in succ(v_i)} P(D[j])] & | \ \exists \overline{v_i} \in V : (v_i, \overline{v_i}) \in C \\ \{v_i\} \cup \bigcup_{j : v_j \in succ(v_i)} D[j] & | \ \text{otherwise} \end{cases} \tag{3.1}$$

The described algorithm can be executed in polynomial time. Per definition, equation (3.1) will be used for each $i \in \{1, \ldots, n\}$ exactly once. As the single dynamic programming cells are calculated in reverse topological order, the values $D[j]$ for all suc-

cessors v_j of a node v_i have already been determined when the value $D[i]$ is calculated. The same holds for each $P(D[j])$. As each node v_i can have at most $|E|$ successors, the complexity for calculating a single value $D[i]$ is $\mathcal{O}(|E|)$. Because this has to be done for each node, the complexity of the complete algorithm is $\mathcal{O}(|E| \cdot |V|)$ in addition to the complexity of calculating the topological order, $\mathcal{O}(|V| + |E|)$.

Note that after $D[1]$ is calculated, we can determine the realization function $r : C \to \mathbb{N}_0$ for realization $G_r = (V_r, E_r)$ such that $V_r = D[1]$ holds by using the following equation:

$$r(c_i) = \begin{cases} l & | \; s_{il} \in D[1] \\ 0 & | \; \forall l \in \{1, \ldots, b_i\} : s_{il} \notin D[1] \end{cases} \tag{3.2}$$

Then, r is a valid realization function as, per equation (3.1), for each conditional pair exactly nodes from one of its branches are selected or none, if the pair is nested into another branch that is not selected.

We now show that for each i, the cell $D[i]$ indeed contains the set of active nodes that maximizes the worst-case execution time of the corresponding sub-problem. For $i = n$, v_i is the last element of the topological order and, as we are considering a DAG with a single sink, $v_i = t$ holds. Because $v_i = t$ per definition does not have any successors, $D[i] = \{v_i\}$ holds per equation (3.1). As the sub-problem of processing $v_i = t$ and every successor consist of solely processing v_i, scheduling the active nodes $\{v_i\}$ maximizes the worst-case execution time of that sub-problem.

Now consider the induction step $i < n$, then per induction hypothesis the values $D[j]$ for all direct successors v_j of v_i denote the set of active nodes that maximize the worst-case execution time of the corresponding sub-problems. If v_i is not the start of a conditional pair, then per definition the subgraphs of all successors of v_i need to be processed. As all non-nested conditional branches are disjoint, processing the union of $\{v_i\}$ and all sets $D[j]$ with $v_j \in succ(v_i)$ that maximize the processing times for their sub-problems, maximizes the worst-case execution time for processing v_i and all successors.

If v_i is the head of a conditional pair, then exactly one conditional branch of that pair needs to be processed. As all non-nested conditional branches are disjoint, selecting the source of the branch with the maximum execution time for its sub-problem maximizes the worst-case execution time of the sub-problem of v_i.

Thus, per equation (3.1) and induction hypothesis, $D[i]$ then contains the set of active nodes that maximizes the worst-case execution time needed to process v_i and all successors. □

The previously shown theorem states that the worst-case execution time of a conditional DAG task on a single machine can be computed in polynomial time. In contrast, Section 2.2 discussed that the computation of the worst-case execution time for a condi-

tional DAG task on a single machine is strongly CoNP-hard if shared nodes are allowed.

In the following, a small example will be used to show that the dynamic program that computes the worst-case execution time for a conditional DAG task on a single machine does not work if shared nodes are allowed.

Therefore, consider the task of Figure 3.1 and assume that jobs 2 and 7 have processing times of two, v_{shared} has a processing time of five and all other jobs have processing times of zero. The algorithm then would for both conditional pairs select the branch that contains v_{shared} and obtain a schedule with an execution time of five. By selecting the upper branch in both conditions, it would be possible to achieve an execution time of seven and thus the algorithm is not optimal in this case.

The polynomial time algorithm and the hardness result for conditional DAGs with shared nodes show, that the independence between different (non-nested) conditions is crucial for the ability to solve the problem in polynomial time for a single machine.

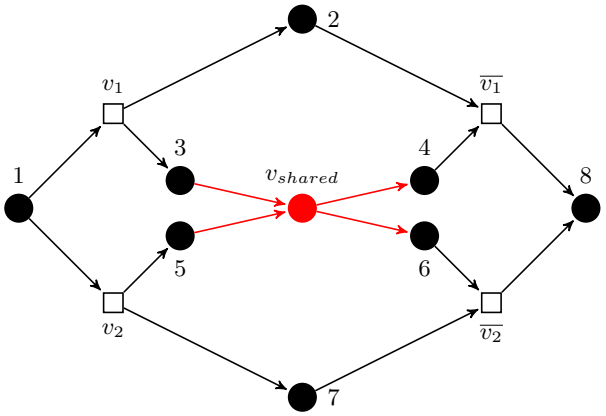


Figure 3.1 Example of a conditional DAG with shared nodes. Nodes and edges that violate the original definition of a conditional DAG are colored in red.

3.1.2 Bounded Width (Pseudo-Polynomial)

In this section, we consider C-DAG for tasks $T = (G, p, \succ, m)$ with the property that the width of each realization G_r of G is bounded by a constant k. The main result of this section is the following theorem.

Theorem 22. C-DAG can be solved exactly in pseudo-polynomial time if each realization G_r of the given conditional DAG G has width at most k.

The basic idea of the pseudo-polynomial time algorithm is inspired by the pseudo-polynomial algorithm for scheduling three chains on two machines in order to minimize the makespan by Agnetis et al. [1]. Their algorithm defines a directed acyclic state graph with weighted edges and a single source s and sink t, such that the length of the shortest s-t-path in the state graph is the minimum makespan for the given scheduling instance.

As that state graph has a pseudo-polynomial number of states, the shortest s-t-path can be computed in pseudo-polynomial time via a dynamic program. The algorithm of this section uses a similar basic idea and defines a state graph with a pseudo-polynomial number of states such that the cumulative weight of the longest s-t-path is the worst-case execution time of the given conditional DAG task. In terms of the state representation the dynamic program uses similar ideas to the algorithm as presented in [31]. Apart from the basic idea, the algorithm will significantly deviate from both mentioned algorithms.

During the course of this section, we assume without loss of generality that the given conditional DAG G has a single source; otherwise we simply add a dummy terminal with execution time zero.

We present a dynamic program (DP) for solving the C-DAG problem. Each state of the DP describes a partial schedule in terms of an *ideal* as defined in [47]. An ideal I of a realization G_r is a subset of V_r such that a job in I implies all of its predecessors to be contained in I as well. We say that I is an ideal of G if I is an ideal of some realization G_r. To respect the precedence constraints and conditions, each partial FP-schedule of G must fulfill the ideal property. Reaching an ideal I' from $I \subset I'$ will correspond to feasibly extending a subschedule by the tasks in $I' \setminus I$ while respecting the FP-order.

An ideal I can be represented in terms of its *front tasks* $I' \subseteq I$, which are all jobs $j \in I$ such that no successor of j is element of I. According to *Dilworth's Decomposition Theorem* [20], an ideal I of a graph G with a width bounded by k can have at most k front tasks. Thus, the number of different ideals is bounded by n^k where n is the number of vertices. A *state* of our dynamic program is a tuple (I, P) with

- $I \subseteq V$ is the set of front tasks of an ideal for some realization G_r of G such that there is a point in time t in the FP-schedule $S_T(G_r)$ where all jobs in I are either being processed or available to being processed.
- For each $j \in I$, $P_j \in \mathbb{N} \cup \{-\}$ either denotes the remaining processing time necessary to complete j or indicates that j has not been started yet ($P_j = -$).

Figure 3.2 exemplary shows a conditional DAG, a partial FP-schedule and the corresponding state representation.

We define a weighted, directed and acyclic state graph $G' = (V', E', w')$ with a distinguished start and end node such that V' contains the states of the dynamic program and E' contains all feasible state transitions. We define G' in such a way that the length of the longest source-sink-path in G' corresponds to the worst-case response time of G. To construct the state graph G', consider the *initial state* $s' = (\{s\}, -)$ where s is the single source of G. The state s' is part of G' and we inductively define the rest of the state graph.

Consider a state $d = (I, P)$ that is part of the state graph. We define A_s as the set of jobs that will be started next according to the FP-order. Let m_d denote the number of jobs $j \in I$ with $P_j \neq -$, i.e., the number of jobs that are being processed in state

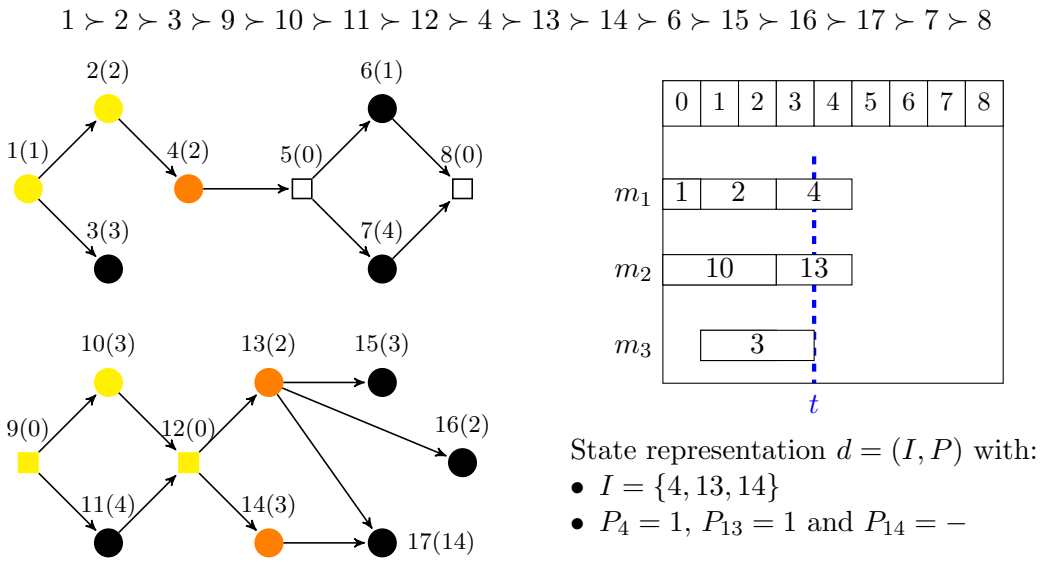


Figure 3.2 Example of a conditional DAG G with processing times annotated in parenthesis behind the corresponding jobs and of an partial fixed-priority schedule of G (top). Additionally shows the state representation of the schedule (bottom). The front tasks of the ideal that represents the schedule are colored in orange and all other jobs of the ideal are colored in yellow.

d. Then $m' = m - m_d$ denotes the number of free machines in d and A_s is the set of the (up-to) m' jobs $j \in I$ with the highest priority and $P_j = -$. If A_s contains jobs with execution times of zero, define A_s to only contain such jobs. This differentiation is necessary because the start of jobs with processing times of zero might cause other jobs to become available. Thus, the set of the m' available jobs with the highest priority might change. Figure 3.4 illustrates the necessity for the differentiation.

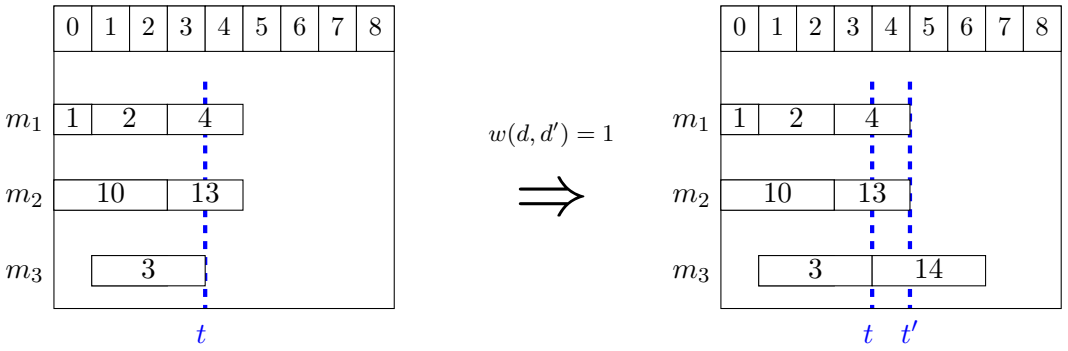
Then, $p_r = \min(\{p_j \mid j \in A_s\} \cup \{P_j \mid j \in I \wedge P_j \neq -\})$ is the time that passes until the next job finishes. Consequently $C = \{j \in A_s \mid p_j = p_r\} \cup \{j \in I \mid P_j = p_r\}$ is the set of jobs that finish next.

We define all states $d' = (I', P')$ and transitions $e = (d, d')$ with $w(e) = p_r$ to be part of G' if the following holds

- $I' = (I \setminus C) \cup S$ contains all jobs $j \in I$ that have not been completed ($j \notin C$). Additionally, I' contains the set of jobs S that become available. Therefore S contains exactly one successor for each $j \in C$ that is the start of a conditional pair and all successors j of other jobs in C for which holds that all predecessors of j have been finished.
- P'_j remains the same for all jobs that were not being processed in d and have not been started, i.e., $P'_j = -$ for all $j \in I \setminus A_s$ with $P_j = -$. The jobs that become available are not being processed, $P'_j = -$ for all $j \in S$.
  For all jobs that have been processed or started in d, but have not finished, the

remaining processing time is decreased by p_r. That is, $P'_j = P_j - p_r$ for all $j \in I$ with $P_j > p_r$ and $P'_j = p_j - p_r$ for all $j \in A_s$ with $p_j > p_r$.

Figure 3.3 exemplary shows a transition starting from state d as defined in the previous example, see Figure 3.2.



State $d = (I, P)$ with:
- $I = \{4, 13, 14\}$
- $P_4 = 1$, $P_{13} = 1$ and $P_{14} = -$

State $d' = (I', P')$ with:
- $I' = \{14, 5, 15, 16\}$
- $P'_{14} = 2$, $P'_{15} = -$, $P'_{16} = -$ and $P'_5 = -$

Figure 3.3 Continuation of the example as illustrated in Figure 3.2. Shows the transition starting from state d.

Note that a state can have multiple successors in G', as multiple sets S of jobs that become available are possible due to the presence of conditional constructs. Figure 3.4 continues the previous example and shows that the state d' has multiple successors.

Additionally, observe that we can decide whether a predecessor j' of a successor j of a completed job has been finished. A predecessor j' has been finished if neither j' nor predecessors of j' are elements of $I \setminus C$.

In summary, the state graph G' contains the start state s' and all states and transitions that can be reached from s' as defined above. Per definition of the state graph, the state $t' = (\emptyset, -)$ is the sink of G'. The following lemma will imply the correctness and completeness of the dynamic program. As the main inductive argument to prove the lemma just exploits that the state transitions formulate each possible extension of a partial fixed-priority schedule for a given conditional DAG G, we will just sketch the proof of the lemma.

Lemma 6. There is an s'-t'-path p in G' with a weight of C if and only if there is a realization G_r of G with an execution time of C.

During the course of the proof sketch, we will use the notion of a state $d = (I, P)$ *representing* point in time t of a schedule S. A state $d = (I, P)$ represents point in time t of schedule S if the following two conditions hold:

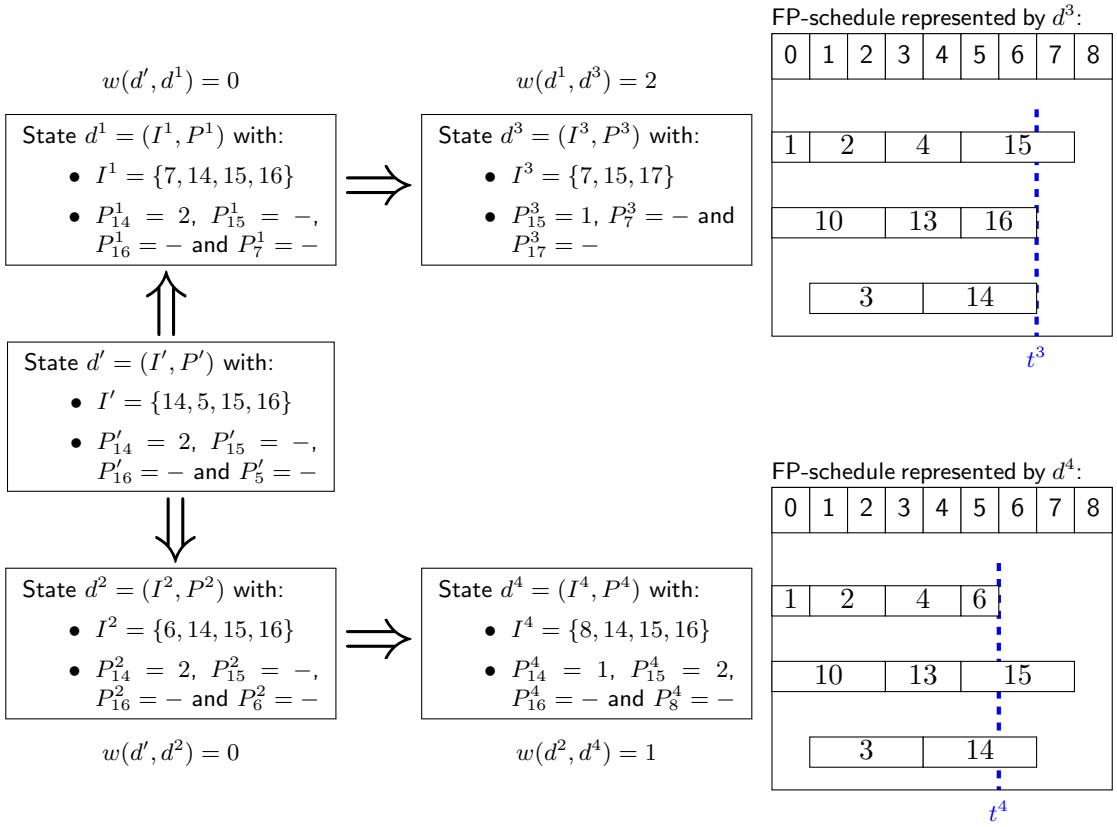


Figure 3.4 Series of transitions starting at state d' as illustrated in Figure 3.3. In state d' the highest priority available job has a processing time of zero and thus only jobs with processing times of zero are started and thus finished. As the completed job 5 is the start of a conditional node, d' has two successor states, d^1 and d^2.

- Each $j \in I$ with $P_j \neq -$ is being processed at point in time t in S with a remaining processing time of P_j and no other jobs are being processed at t in S.
- Each $j \in I$ with $P_j = -$ is available at point in time t in S but not being processed. No other jobs are available at t in S.

Note that multiple states d can represent the same point in time t in S because of jobs with processing times of zero. We will now sketch the proof of the lemma.

Proof Sketch. Let $T = (G, p, \succ, m)$ be the given C-DAG instance and let G' be the state graph for the instance. To show the lemma, we show the following two statements that imply the lemma:

1. If there is a path p from s' to some state d in G' with weight $w(p)$, then there is a realization G_r such that state d represents point in time $w(p)$ in the fixed-priority schedule $S_T(G_r)$, whereas at least one job starts or completes at $w(p)$ in $S_T(G_r)$.
2. For each state d that represents a point in time t in a schedule $S_T(G_r)$ at which a job finishes or starts, there is a path p from s' to d in G' of weight $w(p) = t$.

Statement 1. Assume that there is an s'-t'-path p in G' such that not every state d on p represents point in time $w(p_d)$ in the fixed-priority schedule of some realization, where $w(p_d)$ denotes the weight of the s'-d-subpath of p. Assume that d is the first such state on path p and observe that then $d \neq s'$ must hold, as s' represents point in time zero of any schedule of G per definition.

Thus, there must be a direct predecessor d' of d on path p, such that d' represents point in time $w(p_{d'})$ in the schedule $S_T(G_{r'})$ of some realization $G_{r'}$. Therefore $w(p_d) = w(p_{d'}) + w(e)$ holds with $e = (d', d)$. As d' represents $w(p_{d'})$ in $S_T(G_{r'})$ per assumption and the transition from d' to d formulates one possible continuation of schedule $S_T(G_{r'})$ at point in time $w(p_{d'})$ by definition, there must be a realization G_r such that d represents point in time $w(p_d) = w(p_{d'}) + w(e)$ in schedule $S_T(G_r)$. As this contradicts the assumption, the statement follows.

Statement 2. Assume that there is a realization G_r and a point in time t at which a job starts or completes in $S_T(G_r)$ such that no state d with an s'-d-path p_d of weight $w(p_d) = t$ exists that represents point in time t of schedule $S_T(G_r)$. Assume that t is the earliest point in time in $S_T(G_r)$ for which no such state exists. Note that s' represents point in time zero of every schedule. Thus, there is a greatest point in time t' with $0 \leq t' \leq t$ at which a job completes in $S_T(G_r)$ that is represented by a state d' with an s'-d'-path $p_{d'}$ of weight $w(p_{d'}) = t'$.

Now d' represents point in time t' of schedule $S_T(G_r)$ and t is the next point in time after t' at which a job completes in $S_T(G_r)$. Therefore, by definition of the transitions and list scheduling, there must be a transition e from d' to some state d that represents point in time t of $S_T(G_r)$ with $w(e) = t - t'$. Thus $w(p_d) = t$ holds. As this contradicts the assumption, the statement follows. The statements 1. and 2. together imply the lemma. □

The construction and lemma imply that the total weight of the longest s'-t'-path in G' corresponds to the worst-case execution time of T. We can compute the worst-case execution time $WCET(T)$ for a given conditional DAG task T as follows: we construct the state graph G' and find a longest path from the start state s' to the end state t'. The corresponding worst-case realization G_r and the corresponding fixed-priority schedule can be found via backtracking.

Observe that each state is of polynomial size and that the successor states of a given state d can be computed in polynomial time. Most important, the number of successors of a state d is at most n^k. To see this, observe that multiple successors only occur if start points of conditional pairs finish. As the width of each realization is bound by k, at most k start points of conditional pairs can finish at a time. Because each such start point has at most n branches, at most n^k possible combinations of those branches and thus successors of d exist.

It remains to show that the number of states in G' is pseudo-polynomial in the input size. This would imply that the longest path in G' can be computed in pseudo-polynomial time (of the original input size) and Theorem 22 follows. Note that G' is a DAG by construction and that the longest path between two nodes in a DAG can be computed with a runtime quadratic in the number of nodes, see for example [55].

Observe that for each state (I, P) holds that $|I| \leq k$ since the width of G is bounded by k and thus no ideal can have more than k front tasks. As $I \subseteq V$, there are $n = |V|$ different possible elements and thus $\mathcal{O}(n^k)$ different values for I. Moreover, there are at most $p_{\max}$ possible elements for P, where $p_{\max} = \max_{j \in V} p_j$. Thus, the number of values for P is $\mathcal{O}(p_{\max}^k)$.

We can conclude that the number of states is $\mathcal{O}(n^k \cdot p_{\max}^k)$ which is pseudo-polynomial in the input as k is constant. Thus, the dynamic program can compute the longest s'-t'-path for G' in $\mathcal{O}(n^{2k} \cdot p_{\max}^{2k})$ and, by Lemma 6, solves the C-DAG problem exactly. This proves Theorem 22.

3.2 Approximation Algorithms

As the NP- respectively CoNP-hardness of $\overline{\text{C-DAG}}$ and C-DAG prevents us from finding polynomial time algorithms for both problems, unless P = NP, this section is dedicated to deriving approximation algorithms for optimization problem variants of the problems. The following definition introduces the optimization problem variant of C-DAG.

Definition 36. In the *deadline minimization problem for conditional DAG tasks* (C-DAG MIN), a conditional DAG task $T = (G, p, \succ, m)$ is given and the goal is to minimize value $D \in \mathbb{N}$ with respect to $\forall r \in \mathcal{R}_G : C_T(G_r) \leq D$, which holds if and only if $WCET(T) \leq D$.

Intuitively, the goal of C-DAG MIN is to find the minimal deadline D such that each realization can be guaranteed to finish within D. As the decision problem variant of C-DAG MIN is strongly CoNP-hard, we can conclude that the minimization problem C-DAG MIN is strongly CoNP-hard as well. Analogous, we define the optimization problem variant of $\overline{\text{C-DAG}}$ as follows.

Definition 37. In the *deadline maximization problem for conditional DAG tasks* (C-DAG MAX), a conditional DAG task $T = (G, p, \succ, m)$ is given and the goal is to maximize value $D \in \mathbb{N}$ with respect to $\exists r \in \mathcal{R}_G : C_T(G_r) \geq D$, which holds if and only if $WCET(T) \geq D$.

The goal of C-DAG MAX is to find the maximal deadline D such that at least the execution time of one realization is greater than or equal to D. Note that in the decision problem $\overline{\text{C-DAG}}$, the goal is to decide whether there exists a realization that has

an execution time *strictly* greater than D. While $\overline{\text{C-DAG}}$ and C-DAG MAX deviate from each other in that respect, we can observe that $\overline{\text{C-DAG}}$ would still be strongly NP-hard if the problem was to decide whether there exists a realization that has an execution time greater or equal to D. Thus, the optimization problem C-DAG MAX is strongly NP-hard. We define C-DAG MAX in this way to be able to exploit symmetry properties to C-DAG MIN as discussed in Section 3.2.1. The rest of this section will derive approximation algorithms as defined in the following.

Definition 38. A k-approximation algorithm for an optimization problem P is a polynomial time algorithm A that for all instances of the problem produces a solution whose value is within a factor of k of the value of an optimal solution OPT. That is:

- If P is a maximization problem, $A(I) \geq \frac{1}{k} \cdot OPT$ holds for all instances I.
- If P is a minimization problem, $A(I) \leq k \cdot OPT$ holds for all instances I.

3.2.1 Symmetry Properties

Before we derive approximation algorithms, we first observe some symmetry properties between both problems that can be exploited when deriving approximations.

First, observe that for a given conditional DAG task T, the optimal solution for both, C-DAG MAX and C-DAG MIN, is the worst-case execution time $WCET(T)$. Furthermore, we can even observe that there is a k-approximation for C-DAG MIN if and only if there is a k-approximation for C-DAG MAX. This statement is formulated and proven in the following theorem.

Theorem 23. There exists a k-approximation for C-DAG MIN if and only if there exists a k-approximation for C-DAG MAX.

Proof. If there is a k-approximation for C-DAG MIN, there exists a polynomial time algorithm $A_{\min}$ that, given a conditional DAG task T, computes a value $A_{\min}(T)$ with

$$OPT \leq A_{\min}(T) \leq k \cdot OPT \Leftrightarrow WCET(T) \leq A_{\min}(T) \leq k \cdot WCET(T) \tag{3.1}$$

We can define algorithm $A_{\max}$ with $A_{\max}(T) = \left\lceil \frac{1}{k} \cdot A_{\min}(T) \right\rceil$. Per equation (3.1), the following two inequalities hold.

$$A_{\max} = \left\lceil \frac{1}{k} \cdot A_{\min}(T) \right\rceil \leq \frac{1}{k} \cdot k \cdot WCET(T) = OPT \tag{3.2}$$

$$A_{\max} = \left\lceil \frac{1}{k} \cdot A_{\min}(T) \right\rceil \geq \frac{1}{k} \cdot WCET(T) = \frac{1}{k} \cdot OPT \tag{3.3}$$

$A_{\max}$ can clearly be computed in polynomial time because $A_{\min}$ can be computed in polynomial time by assumption. Additionally, $A_{\max}(T)$ is a feasible solution for C-DAG MAX given T because $D \leq WCET(T)$ is the only constraint on a solution D for

C-DAG MAX by definition and $A_{\max}(T) \leq WCET(T)$ holds per equation (3.2). As, per equation (3.3), $A_{\max}$ also has the necessary approximation factor, it follows that $A_{\max}$ is a k-approximation for C-DAG MAX.

The other direction can be shown analogous. Given a k-approximation $A_{\max}$ for C-DAG MAX, we can construct algorithm $A_{\min}$ with $A_{\min}(T) = \lceil k \cdot A_{\max}(T) \rceil$ and afterwards show that $A_{\min}$ is a k-approximation for $A_{\min}$ similar to the proof above. □

Using this theorem, it is sufficient to find an approximation for one of both problems and the proof of Theorem 23 gives the approximation for the other one.

3.2.2 General Case: 2-Approximation

In this section, a 2-approximation for both optimization problems will be discussed. Therefore, several bounds on the optimal solution for both problems, i.e. $WCET(T)$, will be derived and used to give approximations for the problems. The derivation of this bounds and approximations is based on the *intra-task interference* analysis as introduced in [46]. Note, that the derivation of the upper and lower bounds for $WCET(T)$ is also similar to the analysis of *Graham's list scheduling* as introduced in [30, 37].

We assume without loss of generality that conditional DAGs have a single source s and sink t. If not, we can simply add dummy terminals with processing times of zero.

Let $WCET(T)$ be the worst-case execution time for an arbitrary conditional DAG task $T = (G, p, \succ, m)$. In the following we derive several bounds on $WCET(T)$.

Let P_{st} be the set of all s-t-paths in G and for each $w \in P_{st}$, define $p(w)$ as the sum of all processing times of nodes on w. Let now $w_{\max}$ be the s-t-path that maximizes $p(w)$ over all $w \in P_{st}$. As $w_{\max}$ is an s-t-path, the jobs of $w_{\max}$ cannot be executed in parallel to each other. Because $w_{\max}$ is active for at least one realization function r, at least one realization cannot have a smaller execution time than the cumulative processing time of $w_{\max}$. Therefore the following bound holds:

$$WCET(T) \geq \max_{w \in P_{st}} p(w) = p(w_{\max}) \tag{3.4}$$

Additionally, for each realization G_r it holds that the execution time $C_T(G_r)$ is at least the average load of the machines. Therefore, $WCET(T)$ is at least the maximum average load over all realizations. Thus, the following bound holds, where $avg_{\max} = \frac{1}{m} \cdot \max_{r \in R_G} \sum_{j \in V_r} p_j$ is the maximum average load over all realizations.

$$WCET(T) \geq \frac{1}{m} \cdot \max_{r \in R_G} \sum_{j \in V_r} p_j = avg_{\max} \tag{3.5}$$

Now, consider the realization G_{r^*} of G with $C_T(G_{r^*}) = WCET(T)$, i.e., the realization that leads to the worst-case execution time. Then, the *critical chain* of r^* can be defined as introduced in [46].

Definition 39. The *critical chain* λ_{r^*} of a realization G_{r^*} for a conditional DAG task $T = (G, p, \succ, m)$ is the chain of nodes in G that leads to the worst-case execution time. Let $S_T(G_{r^*})$ be the FP-schedule of G_{r^*}, then the critical chain is defined as follows:

1. The sink t is on the critical chain λ_{r^*}.
2. If node j is on the critical chain, then j' with $(j', j) \in E_{r^*}$ and $C_{j'} = \max_{j'':(j'',j)\in E_{r^*}} C_{j''}$ is on the critical chain, where C_j denotes the completion time of j in schedule $S_T(G_{r^*})$.

We now can describe the worst-case execution time of T as $WCET(T) = p(\lambda_{r^*}) + I_{\lambda_{r^*}}$, where $I_{\lambda_{r^*}}$ denotes the number of time units in which a job of the critical chain is available but not processed in $S_T(G_{r^*})$. Note that, by definition of λ_{r^*}, whenever no job of the critical chain is being processed in $S_T(G_{r^*})$, either a job of λ_{r^*} is available or all jobs have been processed. Thus, the following equation holds:

$$WCET(T) = p(\lambda_{r^*}) + I_{\lambda_{r^*}} \tag{3.6}$$

As λ_{r^*} is an s-t-path, $p(\lambda_{r^*}) \leq p(w_{\max})$ follows. Additionally we can observe that, whenever a job of λ_{r^*} is available but not being processed, all machines must be busy scheduling other jobs by definition of list scheduling. The maximum amount of time all machines can be busy while not processing jobs of λ_{r^*} is $I_{\lambda_{r^*}} \leq avg_{\max} - \frac{1}{m} \cdot p(\lambda_{r^*})$. Therefore, using equation (3.6) and $m \geq 1$, the following bound follows:

$$WCET(T) \leq p(\lambda_{r^*}) + avg_{\max} - \frac{1}{m}p(\lambda_{r^*}) \leq p(w_{\max}) + avg_{\max} - \frac{1}{m}p(w_{\max}) \tag{3.7}$$

Finally, we can exploit that both $avg_{\max}$ and $p(w_{\max})$ are lower bounds on $WCET(T)$, see equations (3.5) and (3.4), to derive the following inequality:

$$WCET(T) \leq p(w_{\max}) + avg_{\max} - \frac{1}{m}p(w_{\max}) \leq (2 - \frac{1}{m})WCET(T) \tag{3.8}$$

This final bound can now be used to show that both, C-DAG MIN and C-DAG MAX can be approximated within a factor of $2 - \frac{1}{m} \leq 2$ in polynomial time.

Theorem 24. C-DAG MIN can be approximated within in factor of $2 - \frac{1}{m}$ in polynomial time.

Proof. Given a C-DAG MIN instance, i.e., a conditional DAG task T, the value $ALG = \left\lfloor p(w_{\max}) + avg_{\max} - \frac{1}{m}p(w_{\max}) \right\rfloor$ is a feasible solution, because $WCET(T) \leq ALG$ holds per equation (3.8) and thus $C_T(G_r) \leq ALG$ holds for all $r \in \mathcal{R}_G$. Additionally, ALG achieves the necessary approximation factor as, by equation (3.8),

$$OPT = WCET(T) \leq ALG \leq (2 - \frac{1}{m})WCET(T) \leq (2 - \frac{1}{m})OPT$$

holds. Therefore, it only remains to show that ALG can be computed in polynomial time and space. In order to do so, the remainder of this proof will argue that $p(w_{\max})$ and $avg_{\max}$ can be computed in polynomial time. It then follows that ALG can be computed in polynomial time as well.

Now, $p(w_{\max})$ is the length of the longest s-t-path in a DAG, which can be found in polynomial time by negating the processing times and searching for the shortest path, as for example described in [55]. For $avg_{\max}$ it holds per definition that $avg_{\max}$ is equal to $\frac{1}{m} \cdot \max_{r \in R_G} \sum_{j \in V_r} p_j$. As $\max_{r \in R_G} \sum_{j \in V_r} p_j$ in turn is the worst-case execution time of T on a single machine and can be computed in polynomial time using the algorithm of Section 3.1.1, $avg_{\max}$ can be computed in polynomial time as well. Thus ALG is a $(2 - \frac{1}{m})$-approximation for C-DAG MIN □

We can exploit the previously shown Theorem 24 and the symmetry of C-DAG MIN and C-DAG MAX to derive the following theorem.

Theorem 25. C-DAG MAX can be approximated within in factor of $2 - \frac{1}{m}$ in polynomial time.

Observe that the approximation relies on computing $\max_{r \in R_G} \sum_{j \in V_r} p_j$ in polynomial time. While this can be done for conditional DAGs, it is not possible for conditional DAGs with shared nodes, unless P = NP. This is because $\max_{r \in R_G} \sum_{j \in V_r} p_j$ is the worst-case execution time on a single machine and computing the worst-case execution time for conditional DAGs with shared nodes is strongly CoNP-hard on a single machine (see Theorem 5). We can conclude that this approximation does not work for conditional DAGs with shared nodes.

We remark that there are publications on schedulability tests for real-time conditional DAG tasks that exploit the bounds of this section to reach the corresponding approximation factor, see for example [11]. While these algorithms guarantee the mentioned approximation factor, they are more complex than a simple computation of the bounds, in order to be more efficient on practical instances. As this thesis only considers provable performance guarantees, we will not discuss these algorithms in detail.

3.2.3 Fully Polynomial-Time Approximation Scheme Preliminaries

After we derived a 2-approximation for the general case of C-DAG MIN respectively C-DAG MAX in the previous section, the goal of the following sections is to find approximations with even better approximation factors for special cases of both problems. In specific, the goal is to derive *fully polynomial-time approximation schemes (FPTAS)* as defined in the following according to the definition given in [28, 60].

Definition 40. A family of algorithms $\{A_\epsilon\}$ is called *fully polynomial-time approximation scheme (FPTAS)* if, for every input I and every $\epsilon > 0$, algorithm A_ϵ finds a solution

of a value within a factor of $1+\epsilon$ (respectively $1-\epsilon$) of the optimal solution for I and the running time of A_ϵ is polynomial in the encoding of I and $\frac{1}{\epsilon}$.

In order to derive fully polynomial-time approximation schemes, we can first observe that for a number of cases that were considered in Chapter 2, there cannot exist an FPTAS, unless P = NP. To see this, consider the following theorem that was shown by Garey and Johnson [28].

Theorem 26. Let P be an optimization problem such that, for all problem instances I, the only possible solution values are positive integers and the optimal solution value OPT is strictly bounded by a polynomial p of the input size and the maximum numeric value in I. Then, if there is a fully polynomial-time approximation scheme $\{A_\epsilon\}$ for P, there also is a pseudo-polynomial time algorithm A for P.

As C-DAG MAX only has positive integer solutions and, according to the bounds that were derived in Section 3.2.2, the optimal solution is appropriately bounded, the requirements of Theorem 26 hold for C-DAG MAX. Now, consider the cases of C-DAG MAX that were shown to be strongly NP-hard in Chapter 2, then we know that there cannot be pseudo-polynomial time algorithms for those cases, unless P = NP. If there was an FPTAS for any of those cases, then per Theorem 26, there must be a pseudo-polynomial time algorithm as well, which can only be the case if P = NP holds. Thus, there cannot be an FPTAS for strongly NP-hard cases of C-DAG MAX, unless P = NP. Because of the symmetry property, see Theorem 23, the same holds for all strongly CoNP-hard cases of C-DAG MIN.

Therefore, we can only hope to find fully polynomial-time approximation schemes for weakly CoNP- respectively NP-hard cases of C-DAG MIN and C-DAG MAX. In the following section we derive an FPTAS for C-DAG MIN for the case of conditional DAGs G, such that each realization G_r of G has a width bounded by a constant k and fulfills a certain monotonicity property. As our only hardness result for bounded width realizations is the weak CoNP-hardness for a constant number of chains, see Theorem 17, an FPTAS for this special case does not contradict previous results.

To derive the FPTAS, the pseudo-polynomial time algorithm of Section 3.1.2 will be used in a rounding approach. The rounding approach is one of the classical approaches to reduce the search space of exact pseudo-polynomial dynamic programs in order to obtain an FPTAS [61]. The rounding approach was for example used by Sahni [52] in the context of scheduling problems.

3.2.4 Fully Polynomial-Time Approximation Scheme under Monotonicity

In this section, we derive an FPTAS for C-DAG MAX and C-DAG MIN for a certain class of conditional DAG tasks. This class contains tasks $T = (G, p, \succ, m)$ with graphs G, such

that each realization G_r has a width bounded by a constant k and is *monotone*, as we define it below. As we will see in Section 3.2.5, this class of graphs contains conditional DAGs G such that each realization G_r is a constant number of disjoint chains.

Intuitively, a scheduling algorithm is *monotone* if increasing the processing times does not decrease the makespan.

Definition 41. A scheduling algorithm is *monotone* for a scheduling instance I, if for any scheduling instance I' that differs from I only in one job j with $p_j < p'_j$, the respective makespans C_I and $C_{I'}$ for each instance satisfy the following:

1. $C_I \leq C_{I'}$, and
2. $C_{I'} \leq C_I + \delta$ with $\delta = p'_j - p_j$.

We call a realization G_r of a conditional DAG G monotone for task $T = (G, p, \succ, m)$ if the fixed-priority schedule using $\succ$ is monotone for $I = (G_r, p, m)$. We say a conditional DAG is monotone for task T, if each realization G_r of G is monotone for T.

The following paragraph gives a family of algorithms $\{A_\epsilon\}$ for C-DAG MIN. Let $T = (G, p, \succ, m)$ be the given C-DAG MIN instance with $G = (V, E, C)$ and $|V| = n$. Let $p_{\max} = \max_{i \in V} p_i$ be the maximum processing time in T. For a fixed $\epsilon > 0$, we let $\mu = \frac{\epsilon \cdot p_{\max}}{n}$ and define algorithm A_ϵ as follows:

1. Let $\hat{T} = (G, \hat{p}, \succ, m)$ be a rounded C-DAG MIN instance with $\hat{p}_j = \left\lceil \frac{p_j}{\mu} \right\rceil$ for each $j \in V$.
2. Solve the rounded instance $\hat{T}$ using the pseudo-polynomial dynamic program as introduced in Section 3.1.2.
3. Let $G_{\hat{r}}$ be the realization corresponding to the longest s'-t'-path p in the state graph as determined by the dynamic program.
4. Return $\hat{C}_{\hat{r}} \cdot \mu$ where $\hat{C}_{\hat{r}}$ is the execution time of $G_{\hat{r}}$ in the rounded instance $\hat{T}$.

The definition of $\{A_\epsilon\}$ follows the standard method of rounding the input values, as for example used in [52], to reduce the state space and thus the running time of the pseudo-polynomial algorithm. We now show that the reduced runtime and the quality of the solutions fulfill the requirements of an FPTAS.

Theorem 27. The family of algorithms $\{A_\epsilon\}$ is a fully polynomial-time approximation scheme for C-DAG MIN for instances $T = (G, p, \succ, m)$ such that the following two conditions hold for each realization G_r of G:

1. The width of G_r is bounded by a constant k.
2. G_r is monotone for T.

Proof. In order to show the theorem, we first argue that the complexity of A_ϵ is polynomial in the input size and $\frac{1}{\epsilon}$ for each fixed $\epsilon > 0$. Let therefore $T = (G, p, \succ, m)$ be an arbitrary input instance that fulfills the requirements of the theorem. Additionally, fix $\epsilon > 0$ arbitrarily.

The values for $p_{\max}$, μ and the rounded processing times $\hat{p}_j$ for each $j \in V$ can be calculated in linear time. Thus, the complexity of A_ϵ is dominated by the complexity of the pseudo-polynomial algorithm. As the algorithm is used on the rounded instance and the width of each realization of G is bounded by a constant k, it has a complexity of $\mathcal{O}(n^{2k} \cdot \hat{p}_{\max}^{2k})$. By definition of the rounded instance it holds that

$$\hat{p}_{\max} \leq \left\lceil \frac{p_{\max}}{\mu} \right\rceil \leq \left\lceil \frac{p_{\max}}{\epsilon \cdot p_{\max}/n} \right\rceil = \left\lceil p_{\max} \cdot \frac{n}{\epsilon p_{\max}} \right\rceil \leq \left\lceil \frac{1}{\epsilon} \cdot n \right\rceil$$

Therefore, the complexity of executing the pseudo-polynomial algorithm is $\mathcal{O}(n^{2k} \cdot \frac{1}{\epsilon}^{2k} \cdot n^{2k}) = \mathcal{O}(n^{4k} \cdot \frac{1}{\epsilon}^{2k})$ and thus the complexity of A_ϵ is polynomial in the input size and $\frac{1}{\epsilon}$.

It remains to show that A_ϵ computes feasible solutions and that the approximation ratio of A_ϵ is not greater than $1 + \epsilon$.

Let $G_{\hat{r}}$ be the realization as determined by A_ϵ and let G_{r^*} be the realization of the optimal solution for the original instance. For each realization G_r of G, let C_r and $\hat{C}_r$ denote the completion times of G_r on the original instance T respectively rounded instance $\hat{T}$. Then, $ALG = \mu \cdot \hat{C}_{\hat{r}}$ is the value calculated by A_ϵ. First, observe that the following inequality holds for each realization G_r:

$$C_r \leq \hat{C}_r \cdot \mu \tag{3.9}$$

To see why this inequality holds, assume that non-integer processing times were allowed and let T^e be a variation of T that uses processing times $p_i^e = \frac{p_i}{\mu}$, i.e., processing times that are scaled down by a factor of exactly μ, and let C_r^e be the completion time of G_r on T^e. Then $C_r = \mu \cdot C_r^e$ holds as the schedules for G_r on the original and exactly scaled instance are identical except for the scaling. Now, compare $\hat{T}$ and T^e, then we can observe that $\hat{T}$ and T^e are identical, except for some jobs that have larger processing times in $\hat{T}$ because of the rounding up. If we show that $\hat{C}_r \geq C_r^e$ holds, the inequality (3.9) follows:

$$C_r = \mu \cdot C_r^e \leq \mu \cdot \hat{C}_r \tag{3.10}$$

As the schedule of each realization is monotone by assumption, the replacement of jobs in the schedule of G_r in instance T^e with longer versions can never decrease the completion time. Therefore $\hat{C}_r \geq C_r^e$ and equation (3.10) follow.

Now, as the pseudo-polynomial algorithm computes the worst-case execution time for the rounded instance, $WCET(\hat{T}) = \hat{C}_{\hat{r}}$ holds and thus, per definition of the worst-case execution time, $\hat{C}_{\hat{r}} \geq \hat{C}_{r^*}$ follows. Therefore, we can derive the following inequality:

$$OPT = C_{r*} \leq \hat{C}_{r^*} \cdot \mu \leq \hat{C}_{\hat{r}} \cdot \mu = ALG \tag{3.11}$$

This inequality already implies that ALG is a feasible solution because $WCET(T) =$

$OPT \leq ALG$ holds by the equation (3.11). Finally, it only remains to show that $ALG \leq (1+\epsilon) \cdot OPT$ holds.

Therefore, again assume that non-integer processing times were allowed and consider instance T^a that uses processing times $p_i^a = \mu \cdot \hat{p}_i$ and let C_r^a be the completion time of G_r in instance T^a. Now, compare C_r^a and $\hat{C}_r$ for an arbitrary realization G_r. As the processing times are scaled by exactly μ and the same realization is considered, $C_r^a = \mu \cdot \hat{C}_r$ holds (this is analogous to the argument on $\mu \cdot C_r^e = C_r$).

Furthermore, compare C_r^a with C_r, then it can be observed that the schedules of C_r^a and C_r use the same jobs except that some jobs in the schedule of C_r^a are longer by at most μ time units due to the rounding. The following two equations show this observation for an arbitrary $j \in V$.

$$p_j^a = \mu \cdot \hat{p}_j = \mu \cdot \left\lceil \frac{p_j}{\mu} \right\rceil \leq p_j + \mu \tag{3.12}$$

$$p_j^a = \mu \cdot \hat{p}_j = \mu \cdot \left\lceil \frac{p_j}{\mu} \right\rceil \geq p_j \tag{3.13}$$

By repeatedly applying condition 2. of the monotonicity for the schedule of G_r, we can conclude that $C_r^a \leq C_r + n \cdot \mu$ must hold because at most n jobs are rounded up and per longer job the execution time can only increase by at most μ time units. Thus, $C_r^a \leq C_r + n \cdot \mu$ follows.

As $C_r^a = \mu \hat{C}_r \leq C_r + n \cdot \mu$ holds for arbitrary G_r, $C_{\hat{r}}^a = \mu \hat{C}_{\hat{r}}$ and $C_{\hat{r}}^a \leq C_{\hat{r}} + n \cdot \mu$ holds in specific. Using this and the definition of $\mu = \frac{\epsilon \cdot p_{\max}}{n}$, the following inequality can be derived

$$ALG = \mu \cdot \hat{C}_{\hat{r}} \leq C_{\hat{r}} + n \cdot \mu = C_{\hat{r}} + \epsilon \cdot p_{\max} \leq OPT + \epsilon \cdot OPT = (1+\epsilon) \cdot OPT$$

where $C_{\hat{r}} \leq WCET(T) = OPT$ holds per definition of the worst-case execution time and $p_{\max} \leq WCET(T) = OPT$ holds because the processing time of the longest job is a lower bound on $WCET(T)$.

Therefore, $OPT \leq ALG \leq (1+\epsilon) \cdot OPT$ holds and thus $\{A_\epsilon\}$ is fully polynomial-time approximation scheme. □

Now, Theorem 27 states that for each $\epsilon > 0$ the algorithm A_ϵ is a $(1+\epsilon)$-approximation for C-DAG MIN if each realization of the given conditional DAG has a bounded width and is monotone. According to the symmetry property, for each $\epsilon > 0$, we can define an algorithm A'_ϵ that is an $(1+\epsilon)$-approximation for C-DAG MAX if each realization of the given conditional DAG has a bounded width and is monotone. As $\frac{1}{1+\epsilon} > 1 - \epsilon$ holds for $\epsilon > 0$, A'_ϵ approximates the solution within a factor of $(1-\epsilon)$.

Therefore, define $A'_\epsilon(T) = \left\lceil \frac{1}{1+\epsilon} \cdot A_\epsilon(T) \right\rceil$, then

$$(1-\epsilon) \cdot WCET(T) \leq \frac{1}{1+\epsilon} \cdot WCET(T) \leq A'_\epsilon(T) \leq WCET(T)$$

holds as stated in Theorem 23. Because the complexity of A_ϵ is polynomial in the input size and $\frac{1}{\epsilon}$, so is the complexity of A'_ϵ. Therefore, we can conclude the following theorem.

Theorem 28. The family of algorithms $\{A'_\epsilon\}$ is a fully polynomial-time approximation scheme for C-DAG MAX for instances $T = (G, p, \succ, m)$ such that the following two conditions hold for each realization G_r of G:

1. The width of G_r is bounded by a constant k.
2. G_r is monotone for T.

3.2.5 Montonicity of Conditional DAGs with Chain Realizations

In the previous section we saw that the introduced families of algorithms $\{A_\epsilon\}$ and $\{A'_\epsilon\}$ are FPTAS for C-DAG MIN respectively C-DAG MAX if each realization of the given conditional DAG has a bounded width and is monotone. In this section, we show that all fixed-priority schedules of conditional DAGs G, such that each realization G_r of G is a constant number of disjoint chains, are monotone. In concrete, the main result of this section is the following theorem.

Theorem 29. Let I be a scheduling instance such that the precedence constraint graph G is a set of disjoint chains, then each list scheduling schedule is monotone for I.

As we are considering realizations G_r, such that the number of chains is bounded by a constant k, the width of each G_r is bounded by k as well. Additionally, Theorem 29 implies that each such realization G_r is also monotone. This observations imply the following theorem.

Theorem 30. The families of algorithms $\{A_\epsilon\}$ and $\{A'_\epsilon\}$ – as introduced in Section 3.2.4 – are fully polynomial-time approximation schemes for C-DAG MIN respectively C-DAG MAX for conditional DAGs G such that each realization G_r is a constant number of chains.

To show Theorem 29, we will separately show that both requirements of monotonicity, see Definition 41, are fulfilled by all list schedules for scheduling instances with precedence constraint graphs that are a set of disjoint chains.

In specific, we will consider two arbitrary scheduling instances $I = (G, p, m)$ and $I' = (G, p', m)$ where G is a set of disjoint chains, $p_i < p'_i$ holds for one job $i \in V$ and $p_j = p'_j$ holds for all $j \neq i$. Let $\succ$ be an arbitrary fixed-priority order over the jobs in V. We will show that $C_j \leq C'_j$ holds for each $j \in V$ where C_j and C'_j denote the

completion times of job j in the list schedules for the instances I and I' using order $\succ$. This statement is formulated in Lemma 8 and is a stronger statement than the first requirement for monotony.

Then, we will show that $C'_j \leq C_j + \delta$ with $\delta = p'_i - p_i$ holds for each $j \in V$ where C_j and C'_j again denote the completion times of job j in the list schedules for the instances I and I' using order $\succ$. This statement is formulated in Lemma 10 and is a stronger statement than the second requirement for monotonicity.

As both, Lemma 8 and 10, are stronger statements than the corresponding requirements for monotonicity, the lemmas then imply Theorem 29 and finally Theorem 30. Before we show the first lemma, we derive an auxiliary lemma. During the rest of this section, we assume without loss of generality that if a job j is scheduled directly after its unique predecessor j', i.e., $(j', j) \in E$ and $S_j = C_{j'}$, then j is scheduled on the same machine as j'.

Lemma 7. Let $I = (G, p, m)$ be a non-conditional scheduling instance such that $G = (V, E)$ is a set of disjoint chains. Let S be a list schedule of I using order $\succ$. If a job j is not scheduled directly after its unique predecessor, i.e., $(j', j) \notin E$ for all j' with $C_{j'} = S_j$, then for all $i \in V$ with $S_i < S_j$ the relation $j \prec i$ holds.

Proof. Let I, S, j and j' be as defined in the lemma. Furthermore, assume that n' is the number of jobs i with $S_i < S_j$. Let $1, \ldots, n'$ be those jobs ordered by starting times, i.e. $S_1 \geq S_2 \geq \ldots \geq S_{n'}$. We show by induction over $i \in \{1, \ldots, n'\}$ that for each job i holds that at S_i another job i' with either $j \prec i'$ or $i' = j$ is available but not started. Then, $i' \prec i$ holds by definition of list scheduling and thus $j \prec i$ holds for each job $i \in \{1, \ldots, n'\}$ and the lemma follows.

Consider case $i = 1$, then at S_i a job i is started. As S_i is the greatest starting time before S_j and j was not scheduled after its unique predecessor by assumption, j must have been available at S_i. Therefore, $i' = j$ is available at S_i but was not scheduled in favor of i. Thus, i has a higher priority than j. Figure 3.5 illustrates this situation.

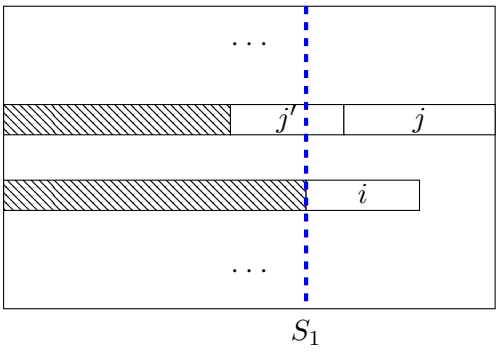


Figure 3.5 Illustration of the induction basis in the proof of Lemma 7, where $(j', j) \notin E$ holds and S_1 is the greatest starting time S_i with $S_i < S_j$. As $(j', j) \notin E$ holds, j must have been available at S_1 and $i \succ j$ follows.

Consider $i+1$ and let l be the job with the smallest starting time such that $S_l > S_{i+1}$ holds. Per induction hypothesis, there is a job l' with either $l' = j$ or $j \prec l'$ that is available at S_l but not scheduled. Now, l has been scheduled instead of l', thus $l' \prec l$ and therefore $j \prec l$ hold.

Let h be the job that finished at S_l on the same processor l is started on. If $(h, l) \in E$, then $(h, l') \notin E$ follows because the precedence constraints are chains and l' must have been available at S_{i+1} but was not scheduled. If $(h, l) \notin E$, then l must have been available at S_{i+1} but was not scheduled. In either way, there exists a job $(i+1)'$ with either $(i+1)' = j$ or $j \prec (i+1)'$ that was available at S_{i+1} but not scheduled. Therefore $(i+1)' \prec i+1$ and thus $j \prec i+1$ must hold. □

We now use Lemma 7 as an auxiliary lemma to show the following lemma, which implies that the first requirement of monotonicity, see Definition 41, holds for all list schedules of scheduling instances whose precedence constraint graphs are a set of disjoint chains.

Lemma 8. Let $I = (G, p, m)$ and $I' = (G, p', m)$ be non-conditional scheduling instances such that $G = (V, E)$ consists of k disjoint chains and with $p_i < p'_i$ for some job $i \in V$ and $p_j = p'_j$ for all $j \neq i$. Let S and S' with makespans $C_{\max}$ and $C'_{\max}$ be the FP-schedules of I and I' for some order $\prec$. Then, $C_j \leq C'_j$ holds for each $j \in V$ where C_j and C'_j denote the completion times of job j in schedule S respectively S'.

Proof. Let I, I', i, S and S' be as described in the lemma. Let S_j and S'_j denote the starting times of a job j in schedule S respectively S'. We will show that $S_j \leq S'_j$ holds for each $j \in V$. The lemma then follows, because, for each $j \in V$, additionally $C_j = S_j + p_j$, $C'_j = S'_j + p'_j$ and $p'_j \geq p_j$ hold per assumption, which implies $C_j \leq C'_j$.

In the following, we assume that there exists at least one job j with $S'_j < S_j$ and show the statement via proof by contradiction. Therefore, assume that job $j+1$ with $S'_{j+1} < S_{j+1}$ is the earliest job in S' with an earlier starting time in S' than in S.

Then, per assumption, for all jobs j with $S'_j \leq S'_{j+1}$ it holds that $S'_j \geq S_j$. This holds especially for the predecessor of $j+1$ in the precedence constraint chain and therefore $j+1$ does not become available earlier in S' than in S. Let R_{j+1} and R'_{j+1} be the point in time at which $j+1$ becomes available in S respectively S', then $R_{j+1} \leq R'_{j+1}$ holds as already stated. If $j+1$ was started directly at R_{j+1} in S, then $S'_{j+1} \geq S_{j+1}$ follows as $j+1$ cannot start earlier than at R'_{j+1} in S' and thus $S'_{j+1} \geq R'_{j+1} \geq R_{j+1} = S_{j+1}$. Figure 3.6 illustrates this situation.

As this contradicts the assumption, assume that $S_{j+1} > R_{j+1}$ holds. From $S_{j+1} > R_{j+1}$ it follows that there is no idle time in interval $[R_{j+1}, S_{j+1}[$ in S as otherwise $j+1$ would have been started earlier by definition of list scheduling. From Lemma 5 it follows that, if there is no idle time in $[R_{j+1}, S_{j+1}[$, there is no idle time in $[0, S_{j+1}[$.

Now, consider the interval $I_1 = [R_{j+1}, S_{j+1}[$ in S, then one can observe that each job

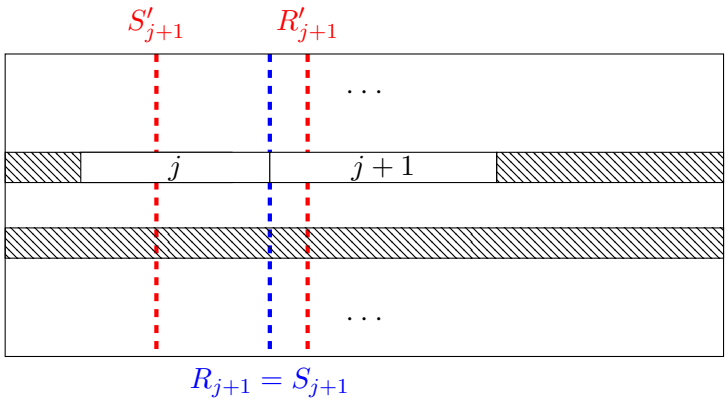


Figure 3.6 Illustration of schedule S as used in the proof of Lemma 8 for the case where $S_{j+1} = R_{j+1}$ holds. Per assumption $S_{j+1} > S'_{j+1}$ holds. But, as $j+1$ is the earliest job in S' that has a earlier starting time in S' than in S, $R'_{j+1} \geq R_{j+1}$ holds and a contradiction ($S'_{j+1} < R'_{j+1}$) occurs.

which is started during that interval must have a higher priority than $j+1$ as otherwise $j+1$ would have been started instead by definition of list scheduling.

Furthermore, consider the interval $I_2 = [0, R_{j+1}[$ in S and specifically the point in time R_{j+1}. As $j+1$ becomes available at R_{j+1}, the predecessor j of $j+1$ must finish at R_{j+1} in S. Because $S_{j+1} > R_{j+1}$ holds, there must be a job j' that is started at R_{j+1} in S instead of $j+1$. This means, that j' must have a higher priority than $j+1$ in the list scheduling order. As j' was not scheduled directly after its unique predecessor, Lemma 7 can be used in order to derive that each job that was started before j' has a higher priority than j' and thus than $j+1$. Figure 3.7 illustrates this situation.

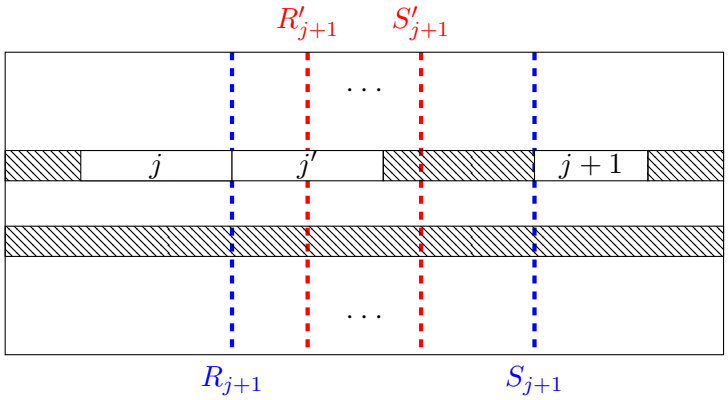


Figure 3.7 Illustration of schedule S as used in the proof of Lemma 8 for the case where $S_{j+1} > R_{j+1}$ holds. $R'_{j+1} \leq S'_{j+1}$ must hold in each feasible schedule, $R'_{j+1} \geq R_{j+1}$ and $S'_{j+1} < S_{j+1}$ hold per assumption. As $(j, j') \notin E$ holds, everything that is started before R_{j+1} has a higher priority than j' and thus $j+1$ per Lemma 7. Everything that starts between R_{j+1} and S_{j+1} has a higher priority than $j+1$ per definition of list scheduling.

Therefore, we can conclude that every job that was started during I_1 or I_2 in S has a

higher priority than $j+1$.

Now, as the assumption at the beginning of the proof was $S'_{j+1} < S_{j+1}$, it follows that $j+1$ must start during interval I_1 or I_2 in S'. Thus, $j+1$ must start in S' before at least some other job that was started during I_1 or I_2 in S. Let J denote the set of jobs that start during I_1 or I_2 in S. As there is no idle time in interval $I = [0, S_{j+1}[$ in S, it means that the last jobs per machine that start during I in S must each be from different chains. Let J_L denote those jobs. If $j+1$ starts after all jobs in J_L in S', it starts outside of interval I in S' because $j+1$ is the earliest job with an earlier starting time in S' than in S. Thus, $j+1$ then starts outside of the intervals I_1 and I_2.

If $j+1$ starts before some job in J_L in S', it follows that there must be a job j' in J that is available but not started when $j+1$ starts. As every job in J has a higher priority than $j+1$, so does j', which is a contradiction to the start of $j+1$. Therefore $S'_{j+1} \geq S_{j+1}$ must hold and the lemma follows. □

As Lemma 8 holds, it directly follows that the first requirement of monotonicity, see Definition 41, holds for all list schedules of scheduling instances whose precedence constraint graphs are a set of disjoint chains. It therefore remains to show that the second requirement of monotonicity is fulfilled as well, as implied by Lemma 10. To do so, we again first proof an auxiliary lemma.

Lemma 9. Let $I = (G, p, m)$ be a non-conditional scheduling instance such that $G = (V, E)$ is a set of disjoint chains. Let S be a list scheduling schedule of I using order $\prec$ and let j be an arbitrary job that starts at S_j. Then for each j' with $S_{j'} > S_j$ it holds that either j' is started directly after its predecessor h, i.e., $(h, j') \in E$ and $C_h = S_{j'}$, or $j' \prec j$.

Proof. Let I, S and j be as described in the lemma. If the number of free machines at S_j is less than or equal to the number of available jobs at S_j, then the number of processors is greater than or equal to the number of chains that still need processing at S_j. Therefore, each remaining job will be scheduled directly after its predecessor finishes and the lemma holds.

On the other hand, if there exists a job j' that is available at S_j but is not started at that point in time, then $j' \prec j$ holds by definition of list scheduling. Let $i_1, \ldots, i_{n'}$ be the jobs that are started after S_j such that $R_{i_k} \neq S_{i_k}$ holds, where R_{i_k} denotes the point in time job i_k becomes available in schedule S.

Assume $S_{i_1} \leq \ldots \leq S_{i_{n'}}$. We show via induction over k that $i_k \prec j$ holds for each i_k. The lemma then follows.

Consider $k = 1$, then i_k is the first job that is started after S_j but not directly after its predecessor. As i_k is the first such job, i_k must have been available at S_j and thus $i_k \prec j$ must hold because otherwise i_k would have been scheduled instead of j.

Consider case $k+1$. Then, all jobs k' that are available at $S_{i_{k+1}}$ with $R_{k'} \neq S_{i_{k+1}}$

either were already available at S_j or became available during $[S_j, S_{i_{k+1}}[$ but were not started up to point in time $S_{i_{k+1}}$. If i_{k+1} was already available at S_j, then $i_{k+1} \prec j$ follows for the same reason as above.

If i_{k+1} became available after S_j but before $S_{i_{k+1}}$, then some job $i_{k'}$ with $k' < k+1$ was scheduled instead of i_{k+1} at $R_{i_{k+1}}$. Therefore $i_{k+1} \prec i_{k'}$ and, per induction hypothesis, $i_{k'} \prec j$ must hold and thus $i_{k+1} \prec j$ and the lemma follow. □

We now use Lemma 9 as an auxiliary lemma to show the following lemma, which implies that the second requirement of monotonicity, see Definition 41, holds for all list schedules of scheduling instances whose precedence constraint graphs are a set of disjoint chains.

Lemma 10. Let $I = (G, p, m)$ and $I' = (G, p', m)$ be non-conditional scheduling instances such that $G = (V, E)$ consists of k disjoint chains and with $p_i < p'_i$ for some job $i \in V$ and $p_j = p'_j$ for all $j \neq i$. Let S and S' with makespans $C_{\max}$ and $C'_{\max}$ be the list scheduling schedules of I and I' for some order $\prec$. Then, $C_j + \delta \geq C'_j$ with $\delta = p'_i - p_i$ holds for each $j \in V$ where C_j and C'_j denote the completion times of job j in schedule S respectively S'.

Proof. Let I, I', i, S and S' be as described in the lemma. Let S_j and S'_j denote the starting times of a job j in schedule S respectively S'. We will show that $S_j + \delta \geq S'_j$ holds for each $j \in V$. The lemma then follows. To see this, observe that $p_j = p'_j$ holds for all $j \neq i$. If additionally $S_j + \delta \geq S'_j$ holds, then $C_j + \delta \geq C'_j$ follows because of $C_j = S_j + p_j$ and $C'_j = S'_j + p'_j$. For job i, we can observe that $S_i = S'_i$ holds because before the start of i, the schedules S and S' are equal by definition. Then, $p_i + \delta = p'_i$ implies $C_i + \delta \geq C'_i$.

To now show that $S_j + \delta \geq S'_j$ holds for each $j \in V$, we assume that there exists at least one job j with $S'_j > S_j + \delta$ and show the statement via proof by contradiction.

Therefore, assume $j+1$ is the earliest job in S' with $S'_{j+1} > S_{j+1} + \delta$. Then, per assumption, for each job j with $S'_j \leq S'_{j+1}$ it holds that $S'_j \leq S_j + \delta$. Additionally, according to Lemma 8, $S_j \leq S'_j$ holds for each job j and thus each job that starts during $I = [0, S'_{j+1}[$ in S' also starts during that interval in S.

We argue that there cannot be any idle time during $[0, S'_{j+1}[$ in S'. Per assumption, it holds that $R'_{j+1} \leq R_{j+1} + \delta \leq S_{j+1} + \delta < S'_{j+1}$, where R_{j+1} and R'_{j+1} denote the point in time at which $j+1$ becomes available in S respectively S', i.e., the point in time the sole predecessor of $j+1$ finishes. As $j+1$ starts at S'_{j+1} in S', there cannot be any idle time in $[R'_{j+1}, S'_{j+1}[$ as otherwise $j+1$ would have been started according to the definition of list scheduling. Thus, per Lemma 5, there also cannot be any idle time in $[0, S'_{j+1}[$ in S'.

Now, consider schedule S. If $p_{j+1} > \delta$ holds, then $t > \delta$ processing time of job $j+1$ is processed during the interval $[S_{j+1}, S'_{j+1}[$ in S as illustrated in Figure 3.8.

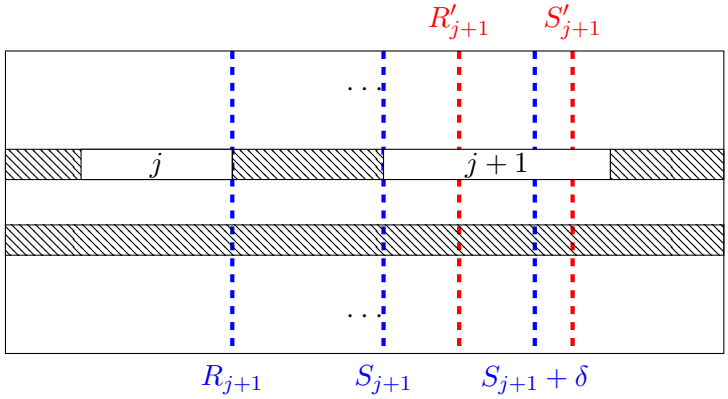


Figure 3.8 Illustration of schedule S as used in the proof of Lemma 10 for the case $p_{j+1} > \delta$. $R_{j+1} \geq S_{j+1}$ holds in each feasible schedule S. $S'_{j+1} > S_{j+1} + \delta$ and $R_{j+1} \leq R'_{j+1} \leq S_{j+1} + \delta$ holds per assumption.

In S' on the other hand, no processing time of $j+1$ is processed in that interval because $j+1$ just starts at S'_{j+1}. As already argued, there cannot be any idle time in $[0, S'_{j+1}[$ in S'. Thus, instead of the t processing time units of $j+1$, processing time of other jobs needs to be processed in S' during that interval instead. Per Lemma 8, no additional job is scheduled during $[0, S'_{j+1}[$ in S' in comparison to S. Additionally, no job is started earlier, so the amount of processing time per job that is processed during $[0, S'_{j+1}[$ cannot be higher in S' than in S. Therefore, the only source of processing time that can replace the t processing time units that moved out of the interval from S to S', is the amount of processing time job i was increased by. As this amount is strictly less than t, idle time must occur, which contradicts the assumption. Figure 3.9 illustrates this argument.

To complete the proof, consider the case where $p_{j+1} \leq \delta$ holds. Let m_1 be the machine on which $j+1$ is scheduled in S, then, as $p_{j+1} \leq \delta$ holds, more jobs must be scheduled on m_1 after $j+1$ during interval $[S_{j+1}, S'_{j+1}[$ in S. If this was not the case, the same contradiction as above would occur. Let A_{j+1} denote the set of jobs that start after $j+1$ during interval $[S_{j+1}, S'_{j+1}[$ in S on machine m_1.

For $j+1$ to start at S'_{j+1} in S', some job in A_{j+1} must be started before $j+1$. If that was not the case, at least $t > \delta$ execution time would have moved out of interval $[0, S'_{j+1}[$ from S to S' and, by the same argument as above, this would lead to a contradiction. If all jobs in A_{j+1} are part of the same chain as $j+1$, all jobs in A_{j+1} must start after $j+1$ and the contradiction occurs. Let therefore $a \in A_{j+1}$ be a job from another chain and assume that a is the first job of its chain in A_{j+1} that is started in S. As a is the first job of its chain in A_{j+1}, it must be scheduled after a job that is not its unique predecessor in S. Therefore, Lemma 7 can be applied in order to conclude that each job that was started before a in S has a higher priority than a, which includes $j+1$.

Then the situation is, that $j+1$ starts at S_{j+1} in S. Additionally, there is a point

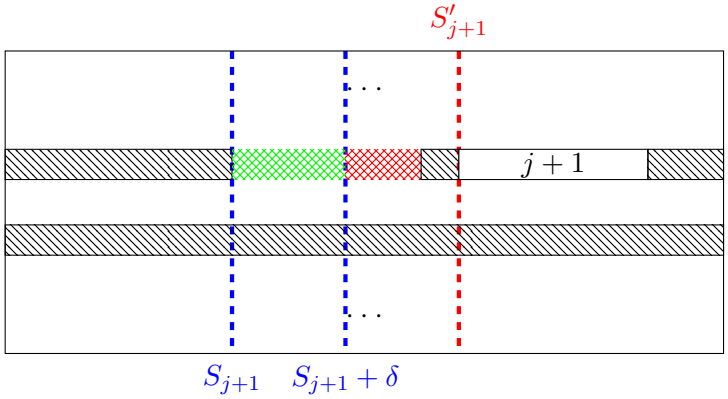


Figure 3.9 Illustration of schedule S' as used in the proof of Lemma 10 for the case $p_{j+1} > \delta$. $S'_{j+1} > S_{j+1} + \delta$ holds per assumption. The area marked with a cross pattern illustrates the amount of processing time that was moved out of interval $[S_{j+1}, S'_{j+1}[$ from S to S', where the green colored area indicates the amount of time that can be replaced by the increase of the processing time of one job by δ. The red colored area indicates the amount of processing time that cannot be replaced.

in time $t = S_{j+1} + \delta$ and finally the point in time S'_{j+1} with $S'_{j+1} > t$ as illustrated in Figure 3.10.

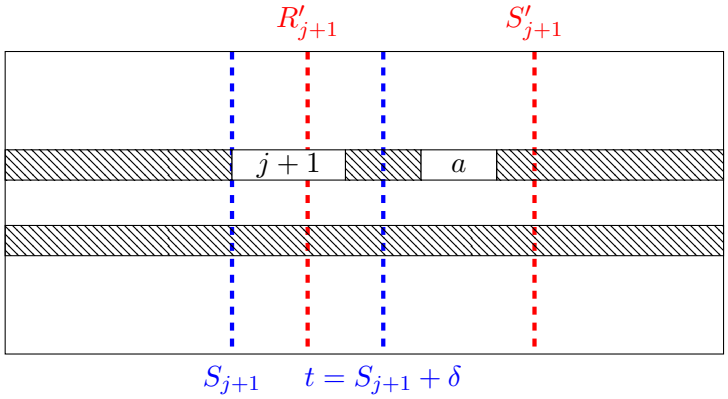


Figure 3.10 Illustration of schedule S as used in the proof of Lemma 10 for the case $p_{j+1} \leq \delta$. $S'_{j+1} > S_{j+1} + \delta$ holds by assumption. Therefore, at least some job a must be executed between $S_{j+1} + \delta$ and S'_{j+1} on the same machine as $j + 1$.

As per assumption $R'_{j+1} \leq R_{j+1} + \delta \leq S_{j+1} + \delta \leq t$ holds, we know that $j + 1$ must be available during $[t, S'_{j+1}]$ in S'. As already argued, a has a lower priority than $j + 1$ and thus, a cannot be started during $[t, S'_{j+1}[$ in S' as $j + 1$ is available and would be scheduled instead by definition of list scheduling. Also, a cannot be scheduled after S'_{j+1}, as otherwise the same contradiction as above would occur.

Therefore, a must be scheduled somewhere in $[C_{j+1}, t[$ in S'. When a is scheduled at S'_a in S', the reason for a to start is, that a has a higher priority than all other available

jobs. Now, according to Lemma 9, each job that is started after a either has a lower priority than a or is started directly after its predecessor finished. Because $j+1$ has a higher priority than a, R'_{j+1} cannot be within $[S'_a, S'_{j+1}[$ in S', because $j+1$ would have to start directly at R'_{j+1}, which contradicts the assumption. Furthermore, R'_{j+1} cannot be smaller than S'_a as otherwise $j+1$ would be available when a is scheduled but has a higher priority, which again would contradict the assumption. And finally, $R'_{j+1} \geq S'_{j+1}$ cannot hold because the schedule would not be feasible. Therefore, every case leads to a contradiction and thus, statement and lemma follow. □

Now, Lemmas 8 and 10 imply Theorem 29, which in turn implies Theorem 30. In summary, we have shown that list schedules for chain precedence constraints are monotone and that therefore the families of algorithms $\{A_\epsilon\}$ and $\{A'_\epsilon\}$ are FPTAS for C-DAG MIN respectively C-DAG MAX if each realization of the given conditional DAG consists of a constant number of disjoint chains.

3.2.6 Montonicity of Conditional DAGs with Bounded Width Realizations

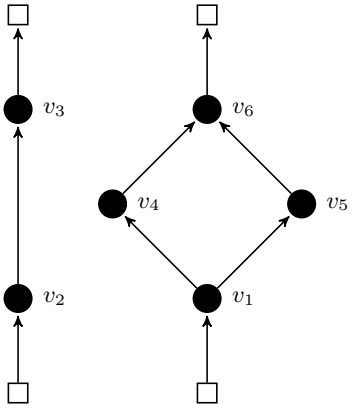
In the previous section, we saw that all list schedules for chain precedence constraints are monotone. In this section we will see that list schedules for scheduling instances with a bounded width are in general not monotone. Therefore, the families of algorithms $\{A_\epsilon\}$ and $\{A'_\epsilon\}$ as introduced in Section 3.2.5 are not FPTAS for all conditional DAGs G, such that each realization of G has a bounded width.

In order to see that instances with a bounded width are not necessarily monotone, consider the following example (Figure 3.11). The example shows a realization G_r of an C-DAG MIN instance $T = (G, p, \succ, 2)$. Additionally, we see two schedules for the example on two machines, for two different processing times of v_2. As the execution time is greater when v_2 has the smaller processing time, the list schedule is not monotone.

When one tries to solve non-monotone conditional DAG tasks with the rounding FPTAS, the algorithm may return infeasible solutions. Consider for example a situation where one tries to solve a task that contains the realization of Figure 3.11 with A_ϵ.

Then, it is possible that $p_1 > p_2$ holds in the original instance, but $\hat{p}_1 = \hat{p}_2$ holds in the rounded instance. Assume that the shown realization leads to the worst-case execution time in the original instance, then the pseudo-polynomial algorithm might find another realization with a higher execution time on the rounded instance. Then $ALG < OPT = WCET(T)$ might hold and ALG is not a feasible solution for C-DAG MIN.

Therefore, the introduced FPTAS does not work for arbitrary conditional DAG tasks whose realizations have a bounded width. Situations like the example of Figure 3.11, where the increase of parameters for a scheduling instance leads to a decrease of its makespan, are sometimes called *Timing Anomalies* and were for example observed and analyzed by L. Graham [30, 37].



Job	v_1	v_2	v_3	v_4	v_5	v_6
Processing time	2	k	4	3	3	4

$$v_1 \succ v_2 \succ v_4 \succ v_5 \succ v_6 \succ v_3$$

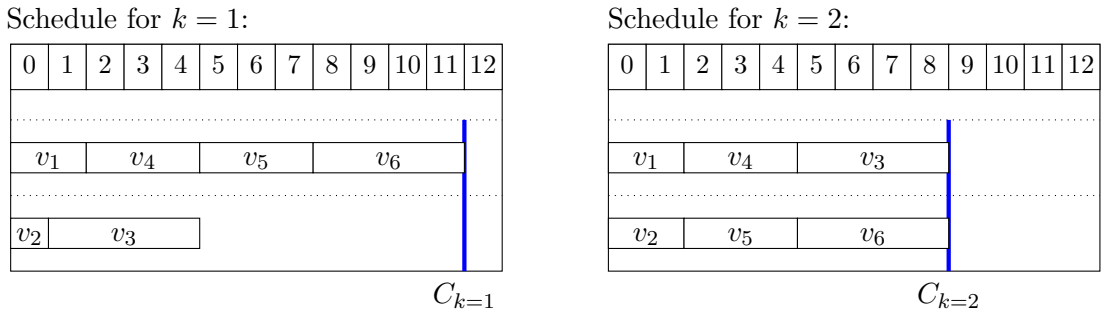


Figure 3.11 The top of the figure shows a realization of width three of a conditional DAG task $T = (G, p, \succ, 2)$, the processing times and a fixed-priority order $\succ$. The bottom shows two different list scheduling schedules induced by $\succ$ for two different processing times of v_2.

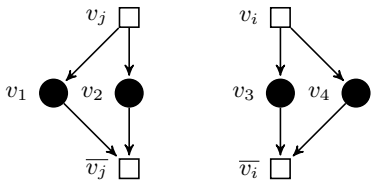
3.3 Restricted Fixed-Priority Orders

This section discusses how the complexity of the worst-case execution time problem for conditional DAG tasks $T = (G, p, \succ, m)$ changes if the priority order $\succ$ cannot be arbitrary but must fulfill some conditions. Therefore, the following definition defines *restricted priority orders*.

Definition 42. Given a conditional DAG $G = (V, E, C)$, a priority order $\succ$ is a *restricted priority order* if for each conditional pair $c_i \in C$ and each pair of nodes $(v, v') \subseteq V_i \times V_i$ holds that no node $v'' \in V \setminus V_i$ exists with $v \succ v'' \succ v'$ respectively $v' \succ v'' \succ v$ where $V_i = \bigcup_{l=1}^{b_i} V_{il}$ is the set of all nodes that are part of any conditional branch of c_i.

Intuitively, a priority order is restricted if each job of the same condition has the "same" priority. To illustrate this definition consider the example in Figure 3.12 that shows a conditional DAG G and two priority orders $\succ_1$ and $\succ_2$.

The order $\succ_1$ is not a restricted priority order according to Definition 42 because v_1 and v_2 both are part of conditional branches of $(v_j, \overline{v_j})$, v_3 is not and $v_1 \succ_1 v_3 \succ_1 v_2$



$$v_j \succ_1 v_1 \succ_1 v_3 \succ_1 v_2 \succ_1 \overline{v_j} \succ_1 v_i \succ_1 v_4 \succ_1 \overline{v_i}$$

$$v_j \succ_2 v_1 \succ_2 v_2 \succ_2 \overline{v_j} \succ_2 v_i \succ_2 v_3 \succ_2 v_4 \succ_2 \overline{v_i}$$

Figure 3.12 Conditional DAG G and two priority orders $\succ_1$ and $\succ_2$ over the nodes of G, where $\succ_2$ is and $\succ_1$ is not a restricted priority order according to Definition 42.

holds. Priority order $\succ_2$ on the other hand is a restricted priority order.

From a practical point of view, conditional DAGs with restricted fixed-priority orders might be interesting. Consider for example a multi-threading system, that just consists of a set of threads that need to be scheduled to complete as fast as possible. In such scenarios, it is a common approach to assign priorities on a thread level. Thus, when modeling each thread with a conditional DAG, each conditional DAG would have the same priority and, in specific, all jobs of the same condition would have the same priority.

Now, consider the class of conditional DAG tasks $T = (G, p, \succ, m)$ with restricted priority orders $\succ$. We can observe that the NP-hardness proofs of Chapter 2 do not show the hardness of the worst-case execution time problem for this class of conditional DAG tasks, because the constructed priority orders do not match Definition 42. In fact, it is possible to show that variants of the worst-case execution time problem, that are NP- respectively CoNP- hard for arbitrary fixed-priority orders, can be solved in polynomial time if the priority order is restricted.

Therefore, consider the class of conditional DAG tasks $T = (G, p, \succ, m)$ for which holds that each realization is monotone and each conditional branch contains just a single job. Note that each reduction in Chapter 2 only constructs conditional branches with a single job. Thus, from the NP-hardness proof in Section 2.4 that shows the hardness of C-DAG for conditional DAGs whose realizations are chains follows that the computation of the worst-case execution time is NP-hard for that class of conditional DAG tasks. Nevertheless, the following theorem states that the problem is solvable in polynomial time if the priority order is restricted.

Theorem 31. *$WCET(T)$ can be computed in polynomial time for conditional DAG tasks $T = (G, p, \succ, m)$ if $\succ$ is a restricted priority order, each realization G_r of G is monotone and $|V_{il}| = 1$ holds for each conditional branch G_{il} with $c_i \in C$ and $l \in \{1, \ldots, b_i\}$.*

Proof. Let $T = (G, p, \succ, m)$ be as described in the theorem, then from $|V_{il}| = 1$ for each conditional branch G_{il} with $c_i \in C$ and $l \in \{1, \ldots, b_i\}$ follows that there cannot be any nested conditional pairs. Let v_{il} denote the single job of branch G_{il} with $l \in \{1, \ldots, b_i\}$

for each $c_i \in C$.

Then, we can define the realization G_r for the conditional DAG G, such that G_r contains the branch with the longest job for each conditional pair c_i. Using the realization, we define the following algorithm ALG:

1. Construct the realization G_r as defined above.
2. Compute the execution time $C_T(G_r)$ of realization G_r
3. Return $ALG = C_T(G_r)$.

As ALG can be executed in polynomial time, it remains to show that $ALG = WCET(T)$ holds. This statement will be shown via proof by contradiction. Therefore assume that some realization G_{r^*} exists with $G_r \neq G_{r^*}$ and $C_T(G_{r^*}) > C_T(G_r)$.

As $G_{r^*} \neq G_r$ holds, at least one c_i exists such that v_{il} is active for G_{r^*}, $v_{il'}$ is active for G_r and $p_{il} < p_{il'}$ holds by definition of G_r. Consider the realization $G_{r'}$ with $V_{r'} = (V_{r^*} \setminus \{v_{il}\}) \cup \{v_{il'}\}$ and $E_{r'} = (E_{r^*} \setminus \{(v_i, v_{il}), (v_{il}, \overline{v_i})\}) \cup \{(v_i, v_{il'}), (v_{il'}, \overline{v_i})\}$. Then G_{r^*} and $G_{r'}$ are equal apart from the replacement of job v_{il} with $v_{il'}$ and the replacement of the incident edges (v_i, v_{il}) and $(v_{il}, \overline{v_i})$ with $(v_i, v_{il'})$ and $(v_{il'}, \overline{v_i})$. As those jobs are part of conditional branches of the same condition, both jobs have the same position in the priority order $\succ$ for the active jobs in G_{r^*} and $G_{r'}$ by definition of restricted priority orders. As this is the case and $p_{il} \leq p_{il'}$ holds by definition of G_r, we can apply the first condition of monotonicity, see Definition 41, to conclude that $C_T(G_{r^*}) \leq C_T(G_{r'})$ holds.

By repeatedly using this swapping argument starting at G_{r^*}, we can derive realizations $G_{r'}$ with $C_T(G_{r'}) \geq C_T(G_{r^*})$, until $G_{r'} = G_r$ and thus $C_T(G_r) \geq C_T(G_{r^*})$ holds. This contradicts the assumption and the theorem follows. □

With Theorem 31 we have seen that there exist classes of conditional DAG tasks for which the worst-case execution time problem is NP- respectively CoNP- hard if the priority order can be arbitrary, but can be solved in polynomial time if the priority order is restricted according to Definition 42.

For now, it remains open whether the general worst-case execution time problem for conditional DAG tasks is still NP-hard if the fixed-priority order is restrictive.

Chapter 4

Conclusion and Outlook

In this thesis, we considered the problem of computing the worst-case execution time for conditional DAGs given a fixed-priority order (C-DAG) and derived several complexity results as well as algorithms for the problem.

First, we discussed the case where the conditional DAG is executed on a single machine. As shown in [46], the computation of the worst-case execution time is independent of the used fixed-priority order, if the conditional DAG is processed on a single machine, and can be done in polynomial time. We were able to show that the independence between (non-nested) conditions is crucial for this result to hold. In specific, we showed that the problem is indeed strongly CoNP-hard for conditional DAGs with shared nodes, i.e., conditional DAGs that allow common nodes between branches of different (non-nested) conditions.

Then, we considered the more usual conditional DAG model, that does not allow common nodes between different (non-nested) conditional branches. We proved that the general worst-case execution time problem for this model is CoNP-hard in the strong sense and remains so even if we only consider two-terminals series-parallel conditional DAGs or allow preemption. During the course of this proof, we exploited a non-obvious relation between the worst-case execution time problem and the list scheduling makespan maximization problem (LS MAX). LS MAX is the problem of finding the maximum makespan of a scheduling instance that can be achieved using a list scheduling algorithm. We then showed the CoNP-hardness of C-DAG by first proving the NP-hardness of LS MAX and then reducing from LS MAX. We were able to observe that the reduction from LS MAX to C-DAG in some way preserves the structure of the input precedence constraint graph into the constructed conditional DAG and exploited this observation to derive a proof framework; in order to show the hardness of C-DAG for special graph classes, it is sufficient to show the hardness of LS MAX for the corresponding graph class.

While we were able to use this proof framework to derive further results, we observed that it depends on the ability to arbitrarily assign priorities to all jobs of a conditional DAG. In specific, the reduction does not work if we require all jobs of the same condition to have the same priority. Therefore, we considered C-DAG for restricted priority orders.

That is, a problem variant that only considers fixed-priority orders that enforce all jobs of a condition to have the same priority. We were able to show that there exist variants of C-DAG that are CoNP-hard for arbitrary priority orders but can be solved in polynomial time if the priority order is restricted. The complexity of the general case of this problem variant remains open and could be considered in future work, because it might be relevant in practice as it is a common approach to assign priorities on a thread level in multithreading scenarios. Thus, when modeling each thread with a conditional DAG, each conditional DAG would have the same priority and, in specific, all jobs of the same condition would have the same priority.

Nevertheless, using the proof framework we were able derive further hardness results for C-DAG for special graph classes. In specific, we were able to show that the problem is still strongly CoNP-hard if each realization of the conditional DAG is a tree and weakly CoNP-hard if each realization is a constant number of disjoint chains. To be more precise, we were able to derive the latter result for $k = 4$ and $m = 2$, where k is the constant number of chains and m is the number of machines used to process the DAG. The complexity of the smallest non-trivial case, $k = 3$ and $m = 2$, remains open and could be discussed in future work. However, we were able to show that the hardness for $k = 3$ and $m = 2$ cannot be derived using the introduced proof framework, because LS MAX is solvable in polynomial time for the corresponding case. Additionally, we were able to derive that the general version of the problem as well as the special case of trees remain weakly CoNP-hard if we fix the number of machines at two. The possibility of a stronger hardness result for a fixed number of machines remains an open question.

Besides the hardness proofs, we also derived positive results. In specific, we introduced a pseudo-polynomial dynamic program that computes the worst-case execution time for conditional DAGs with realizations that have a bounded width. We were then able to use the dynamic program in a rounding approach to derive an FPTAS for the case of conditional DAGs with realizations that have a bounded width and fulfill a monotonicity condition. The monotonicity condition formulates that an increased processing time for a single job cannot lead to a decreased makespan in the fixed-priority schedules. While this condition does not hold in general [30, 37], we were able to show that each realization that is a constant number of disjoint chains is monotone. Therefore, the rounding approach is also an FPTAS for conditional DAGs with realizations that are a constant number of disjoint chains. Additionally, we discussed the existing 2-approximation for the general case, see [46]. We remark that this algorithm again relies on the ability to compute the worst-case execution time for a single machine in polynomial time and thus does not work for the case of conditional DAGs with shared nodes.

One motivation for this thesis was the relevance of conditional DAGs and fixed-priority scheduling in the context of real-time scheduling and sporadic tasks. While our hardness results have direct relevance for the problem of schedulability testing sporadic conditional

DAG tasks, it remains open whether our newly derived algorithms – in specific the pseudo-polynomial time algorithm for conditional DAGs with bounded width realizations and the FPTAS for conditional DAGs with monotone bounded width realizations – could be exploited in that context as well.

Bibliography

[1] A. Agnetis, M. Flamini, G. Nicosia, and A. Pacifici. "Scheduling three chains on two parallel machines". In: *European Journal of Operational Research* 202.3 (2010), pp. 669–674.

[2] F. E. Allen. "Control Flow Analysis". In: *SIGPLAN Not.* 5.7 (July 1970), pp. 1–19.

[3] B. Andersson and D. de Niz. "Analyzing Global-EDF for Multiprocessor Scheduling of Parallel Tasks". In: *Principles of Distributed Systems: 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012*. Dec. 2012, pp. 16–30.

[4] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. 1st. New York, NY, USA: Cambridge University Press, 2009.

[5] N. Audsley, A. Burns, M. Richardson, and A. Wellings. "Applying new Scheduling Theory to Static Priority Pre-emptive Scheduling". In: *Software Engineering Journal* 8.5 (1993), pp. 284–292.

[6] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Döbel, and H. Härtig. "Response-Time Analysis of Parallel Fork-Join Workloads with Real-Time Constraints". In: *2013 25th Euromicro Conference on Real-Time Systems*. July 2013, pp. 215–224.

[7] S. Baruah. "Improved Multiprocessor Global Schedulability Analysis of Sporadic DAG Task Systems". In: *2014 26th Euromicro Conference on Real-Time Systems*. July 2014, pp. 97–105.

[8] S. Baruah. "The federated scheduling of systems of conditional sporadic DAG tasks". In: *2015 International Conference on Embedded Software (EMSOFT)*. Oct. 2015, pp. 1–10.

[9] S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela. "The Global EDF Scheduling of Systems of Conditional Sporadic DAG Tasks". In: *2015 27th Euromicro Conference on Real-Time Systems*. July 2015, pp. 222–231.

[10] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese. "A Generalized Parallel Task Model for Recurrent Real-time Processes". In: *2012 IEEE 33rd Real-Time Systems Symposium*. Dec. 2012, pp. 63–72.

[11] S. Baruah. "Resource-Efficient Execution of Conditional Parallel Real-Time Tasks". In: *Euro-Par 2018: Parallel Processing*. Ed. by M. Aldinucci, L. Padovani, and M. Torquati. Cham: Springer International Publishing, 2018, pp. 218–231.

[12] S. K. Baruah. “Dynamic- and Static-priority Scheduling of Recurring Real-time Tasks”. In: *Real-Time Systems* 24.1 (Jan. 2003), pp. 93–128.

[13] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese. “Feasibility Analysis in the Sporadic DAG Task Model”. In: *2013 25th Euromicro Conference on Real-Time Systems*. July 2013, pp. 225–233.

[14] V. Bonifaci, A. Marchetti-Spaccamela, N. Megow, and A. Wiese. “Polynomial-Time Exact Schedulability Tests for Harmonic Real-Time Tasks”. In: *RTSS*. IEEE Computer Society, 2013, pp. 236–245.

[15] S. Chakraborty, T. Erlebach, S. Kunzli, and L. Thiele. “Schedulability of event-driven code blocks in real-time embedded systems”. In: *Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324)*. June 2002, pp. 616–621.

[16] S. Chakraborty, T. Erlebach, and L. Thiele. “On the Complexity of Scheduling Conditional Real-Time Code”. In: *Algorithms and Data Structures*. Ed. by F. Dehne, J. Sack, and R. Tamassia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 38–49.

[17] H. S. Chwa, J. Lee, K. Phan, A. Easwaran, and I. Shin. “Global EDF Schedulability Analysis for Synchronous Parallel Tasks on Multicore Platforms”. In: *2013 25th Euromicro Conference on Real-Time Systems*. July 2013, pp. 25–34.

[18] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. 1st. Secaucus, NJ, USA: Birkhauser Boston, Inc., 2000.

[19] R. Diestel. *Graph theory*. 4. ed. Graduate Texts in Mathematics ; 173. Heidelberg [u.a.]: Springer, 2010.

[20] R. P. Dilworth. “A Decomposition Theorem for Partially Ordered Sets”. In: *Annals of Mathematics* 51.1 (1950), pp. 161–166.

[21] J. Du, J. Y.-T. Leung, and G. H. Young. “Scheduling chain-structured tasks to minimize makespan and mean flow time”. In: *Information and Computation* 92.2 (1991), pp. 219–236.

[22] F. Eisenbrand and T. Rothvoß. “Static-Priority Real-Time Scheduling: Response Time Computation Is NP-Hard”. In: *2008 Real-Time Systems Symposium*. Nov. 2008, pp. 397–406.

[23] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. “Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE ’98. Le Palais des Congrès de Paris, France: IEEE Computer Society, 1998, pp. 132–139.

[24] H. Elrewini and H. H. Ali. “Static scheduling of conditional branches in parallel programs”. In: *Journal of parallel and Distributed computing* 24.1 (1995), pp. 41–54.

[25] D. Eppstein. “Parallel recognition of series-parallel graphs”. In: *Information and Computation* 98.1 (1992), pp. 41–55.

[26] J. C. Fonseca, V. Nélis, G. Raravi, and L. M. Pinho. “A multi-DAG Model for Real-time Parallel Applications with Conditional Execution”. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC ’15. Salamanca, Spain: ACM, 2015, pp. 1925–1932.

[27] J. Fonseca, G. Nelissen, and V. Nélis. “Schedulability analysis of DAG tasks with arbitrary deadlines under global fixed-priority scheduling”. In: *Real-Time Systems* (Feb. 2019).

[28] M. R. Garey and D. S. Johnson. ““ Strong ” NP-Completeness Results: Motivation, Examples, and Implications”. In: *J. ACM* 25.3 (July 1978), pp. 499–508.

[29] O. Goldreich. *Computational Complexity: A Conceptual Perspective*. 1st ed. New York, NY, USA: Cambridge University Press, 2008.

[30] R. L. Graham. “Bounds for Certain Multiprocessing Anomalies”. In: *Bell System Technical Journal* 45.9 (1966), pp. 1563–1581.

[31] E. Günther, F. G. König, and N. Megow. “Scheduling and packing malleable and parallel tasks with precedence constraints of bounded width”. In: *Journal of Combinatorial Optimization* 27.1 (Jan. 2014), pp. 164–181.

[32] T. C. Hu. “Parallel Sequencing and Assembly Line Problems”. In: *Oper. Res.* 9.6 (Dec. 1961), pp. 841–848.

[33] K. Jansen and R. Solis-Oba. “Approximation Algorithms for Scheduling Jobs with Chain Precedence Constraints”. In: *Parallel Processing and Applied Mathematics*. Ed. by R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Waśniewski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 105–112.

[34] M. Joseph and P. K. Pandya. “Finding Response Times in a Real-Time System”. In: *The Computer Journal* 29.5 (1986), pp. 390–395.

[35] A. B. Kahn. “Topological Sorting of Large Networks”. In: *Commun. ACM* 5.11 (Nov. 1962), pp. 558–562.

[36] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. “Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car”. In: *2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*. Apr. 2013, pp. 31–40.

[37] R. L. Graham. “Bounds on Multiprocessing Timing Anomalies”. In: *Siam Journal on Applied Mathematics - SIAMAM* 17 (Mar. 1969).

[38] K. Lakshmanan, S. Kato, and R. Rajkumar. “Scheduling Parallel Real-Time Tasks on Multi-core Processors”. In: *2010 31st IEEE Real-Time Systems Symposium*. Nov. 2010, pp. 259–268.

[39] J. Lenstra, A. R. Kan, and P. Brucker. “Complexity of Machine Scheduling Problems”. In: *Studies in Integer Programming*. Ed. by P. Hammer, E. Johnson, B. Korte, and G. Nemhauser. Vol. 1. Annals of Discrete Mathematics. Elsevier, 1977, pp. 343–362.

[40] J. Li, K. Agrawal, C. Lu, and C. Gill. "Outstanding Paper Award: Analysis of Global EDF for Parallel Tasks". In: *2013 25th Euromicro Conference on Real-Time Systems*. July 2013, pp. 3–13.

[41] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah. "Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks". In: *2014 26th Euromicro Conference on Real-Time Systems*. July 2014, pp. 85–96.

[42] C. L. Liu and J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* 20 (1973), pp. 46–61.

[43] M. Lombardi and M. Milano. "Allocation and scheduling of Conditional Task Graphs". In: *Artificial Intelligence* 174.7 (2010), pp. 500–529.

[44] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho. "Response-Time Analysis of Synchronous Parallel Tasks in Multiprocessor Systems". In: *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*. RTNS '14. Versaille, France: ACM, 2014, 3:3–3:12.

[45] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo. "Schedulability Analysis of Conditional Parallel Task Graphs in Multicore Systems". In: *IEEE Transactions on Computers* 66.2 (Feb. 2017), pp. 339–353.

[46] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo. "Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems". In: *2015 27th Euromicro Conference on Real-Time Systems*. July 2015, pp. 211–221.

[47] R. H. Möhring. "Computationally Tractable Classes of Ordered Sets". In: *Algorithms and Order*. Ed. by I. Rival. Dordrecht: Springer Netherlands, 1989, pp. 105–193.

[48] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. "Techniques Optimizing the Number of Processors to Schedule Multi-threaded Tasks". In: *2012 24th Euromicro Conference on Real-Time Systems*. July 2012, pp. 321–330.

[49] A. Parri, A. Biondi, and M. Marinoni. "Response Time Analysis for G-EDF and G-DM Scheduling of Sporadic DAG-tasks with Arbitrary Deadline". In: *Proceedings of the 23rd In ternational Conference on Real Time and Networks Systems*. RTNS '15. Lille, France: ACM, 2015, pp. 205–214.

[50] M. L. Pinedo. *Scheduling: theory, algorithms, and systems*. 4. ed. New York [u.a.]: Springer, 2012.

[51] M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet. "Global EDF Scheduling of Directed Acyclic Graphs on Multiprocessor Systems". In: *Proceedings of the 21st International Conference on Real-Time Networks and Systems*. RTNS '13. Sophia Antipolis, France: ACM, 2013, pp. 287–296.

[52] S. K. Sahni. "Algorithms for Scheduling Independent Tasks". In: *J. ACM* 23.1 (Jan. 1976), pp. 116–127.

[53] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. “Multi-core Real-Time Scheduling for Generalized Parallel Task Models”. In: *2011 IEEE 32nd Real-Time Systems Symposium*. Nov. 2011, pp. 217–226.

[54] T. J. Schaefer. “The Complexity of Satisfiability Problems”. In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*. STOC ’78. San Diego, California, USA: ACM, 1978, pp. 216–226.

[55] R. Sedgewick and K. Wayne. *Algorithms*. 4. ed. Upper Saddle River, NJ [u.a.]: Addison-Wesley, 2011.

[56] M. A. Serrano, A. Melani, M. Bertogna, and E. Quinones. “Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions”. In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2016, pp. 1066–1071.

[57] K. W. Tindell, A. Burns, and A. J. Wellings. “An extendible approach for analyzing fixed priority hard real-time tasks”. In: *Real-Time Systems* 6.2 (Mar. 1994), pp. 133–151.

[58] J. Ullman. “NP-complete scheduling problems”. In: *Journal of Computer and System Sciences* 10.3 (1975), pp. 384–393.

[59] J. Valdes Ayesta. “Parsing Flowcharts and Series-parallel Graphs.” AAI7905944. PhD thesis. Stanford, CA, USA: Stanford University, 1978.

[60] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*. 1st. New York, NY, USA: Cambridge University Press, 2011.

[61] G. J. Woeginger. “When Does a Dynamic Programming Formulation Guarantee the Existence of a Fully Polynomial Time Approximation Scheme (FPTAS)?” In: *INFORMS Journal on Computing* 12.1 (2000), pp. 57–74.