Conditional Scheduling with Varying Deadlines

Benjamin Horowitz
bhorowit@cs.berkeley.edu

Report No. UCB/CSD-02-1220

December 13, 2002

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Abstract

We examine a conditional scheduling model in which the deadlines of jobs are allowed to vary. We explore variants of the basic model and locate the dividing line between NP-hard and polynomial-time problems.

Suppose that an engineer needs to design one of two devices, either A or B. Each device will require nine months of work. The due date of both A and B is in one year, but only one device will need to be completed. However, which device will need to be built will not be known until six months from now. Building both devices is clearly impossible, since the total amount of time required is 18 months. Instead, the engineer's next six months should be divided between A and B, with the remaining six months devoted exclusively to one or the other, depending on the decision six months from now about which device to complete. In this example, the scheduling strategy is easy enough to determine. In more complicated examples, it may be less clear how to devise a strategy. What if the devices to be made are many, require unequal amounts of time, and have unequal due dates? What if the decisions about which device to make occur at different times? What if some devices require other devices to be already completed? This report presents generalizations and variants of the engineer's problem, and locates the dividing line between polynomial-time and computationally infeasible variants.

1 Related work

The model we shall consider is closely related to the model of [Bar98a], which was further extended and analyzed in [Bar98b, CET01]. Our model generalizes these models in two respects. First, and most importantly, the models of [Bar98a, Bar98b, CET01] do not permit the deadlines of jobs to change depending on conditional behavior. Though the idea of dynamically changing deadlines may seem odd, we have already encountered a situation in which the deadlines do so change: the engineer's problem. In one scenario, the engineer had to produce device A within 12 months, with device B given an infinite deadline. In the other scenario, then engineer had to produce device B within 12 months, with device A given an infinite deadline. Second, jobs are unrelated in the models of [Bar98a, Bar98b, CET01], in the sense that jobs do not admit precedence constraints. When precedence constraints are included, conditionally changing deadlines are also required, even if the relative deadline of each job j is fixed when j is released. To see why this is so, consider a scenario in which job j_1 precedes both j_2 and j_3. Initially, j_1 is released, followed some fixed time later by job either j_2 or j_3, but not both. If j_2's deadline is different from j_3's deadline, then depending on which branch is followed, j_1's deadline varies.

Though the model we shall consider generalizes those of [Bar98a, Bar98b, CET01] in the two respects just discussed, in other respects our model is weaker. We do not address the parallel composition of two or more conditional scheduling problems. This is in contrast to [Bar98a, Bar98b], though the less vague analysis of [CET01] also does not address compositionality. Our techniques could be extended to handle such compositions, though we do not discuss these extensions here. Second, our models specify exact release times for tasks, not minimum separations between release times. Finally, unlike [CET01], we do not discuss approximation algorithms for those problems that we prove to be hard (NP-hard or otherwise). Despite these modest restrictions, we believe our scheduling algorithms are novel and of independent interest.[1]

[1]The reader familiar with the techniques of [Bar98a, Bar98b, CET01] may wonder whether these techniques may be extended to our setting. We now explain why such an extension seems improbable. These techniques rely on a *deadline bound function*, which assigns to each nonnegative real number t a number $dbf(t)$, which is the maximum, over all time intervals I of length t, of the sum of computation times of jobs that have release times and deadlines within I. As mentioned above, in the models of [Bar98a, Bar98b, CET01], each job j is due at its release time plus a fixed constant $d(j)$. It is shown in [CET01] that an instance of such a model is schedulable if and only if for all $t \geq 0$,

$$dbf(t) \leq t \tag{1.0.1}$$

The proof of the "only if" portion relies in an essential way on the optimality of the earliest deadline first (EDF) algorithm. However, deadlines vary in our setting, and it is therefore unclear what an EDF algorithm would be. Thus, the proof of condition (1.0.1) does not extend to our setting. Indeed, it is easy to see that (1.0.1) is not sufficient in our setting. (It remains necessary, though we do not discuss its necessity here.) Consider the example from the beginning of the report, modified so that both device A and device B require 10 months of time. Then

$$dbf(t) = \begin{cases} 0 & \text{if} \quad t < 12 \\ 10 & \text{if} \quad t \geq 12 \end{cases}$$

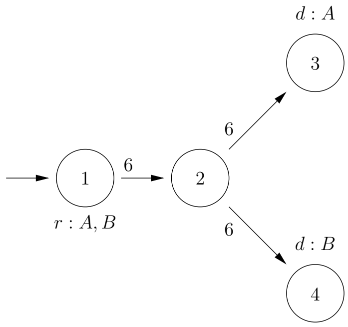

Figure 2.0.1: The engineer's problem modeled as a conditional scheduling problem.

Another line of research which has influenced our model is that of G. Fohler, especially [Foh94], which develops algorithms for multiprocessor conditional scheduling. In contrast to [Foh94], our focus is on polynomial-time algorithms. In addition, our algorithms are optimal: they find a feasible schedule if one exists. Because of our focus on efficiency and optimality, we avoid multiprocessor scheduling problems. Approximation algorithms and heuristics for multiprocessor conditional scheduling may also be found, for example, in [GS98, EKP$^+$98].

A third line of research in real-time scheduling seeks to extend the priority ceiling protocol [SRL90] to handle changes of operational mode, including the addition or deletion of tasks, modification of the frequency of a task, and so on [SRLR89]. Our approach differs substantially from that of [SRLR89]. In our model, jobs interact via precedence constraints, not via shared resources. Moreover, mode changes seem to be regarded as infrequent events in [SRLR89], whereas in our model, mode changes are allowed to be frequent. Finally, as noted above, our schedulability tests are sufficient and necessary, whereas [SRLR89] only provides sufficient tests.

2 The conditional scheduling problem

We begin with a precise definition of the model we shall consider in this report.

Definition 2.0.1 (conditional scheduling problem). A *conditional scheduling problem* P is a pair $(\mathcal{F}, W)$, where:

- $\mathcal{F} = (\mathcal{V}, v_0, \mathcal{E}, \mathcal{D})$ is called the *finite state machine of* P. Here, $\mathcal{V}$ is a finite set, called the *vertices*, of which the *initial vertex* $v_0 \in \mathcal{V}$ is a member. The set $\mathcal{E}$ of *edges* is a subset of $\mathcal{V} \times \mathcal{V}$; we write $v \to v'$ instead of $(v, v') \in \mathcal{E}$. The function $\mathcal{D}$ assigns to each edge $e \in \mathcal{E}$ a positive rational *duration* $\mathcal{D}(e) \in \mathbb{Q}^{>0}$.
- $W = (J, t, r, d)$ is called the *workload of* P. Here, J is a finite set, called the *jobs*. The function t assigns to each job $j \in J$ an amount $t(j) \in \mathbb{Q}^{>0}$ of *time* required by j. The function r assigns to each vertex $v \in \mathcal{V}$ a set $r(v) \subseteq J$ of jobs *released* at v. Similarly, the function d assigns to each vertex $v \in \mathcal{V}$ a set $r(v) \subseteq J$ of jobs *due* at v. $\square$

Example 2.0.2. Figure 2.0.1 presents the engineer's problem modeled as a conditional scheduling problem. For convenience, we use integers as vertices. At the initial vertex $v_0 = 1$, jobs A and B are released. The jobs released at vertex 1 are indicated with the label $r : A, B$ adjacent to vertex 1. The time required by each job A and B is 9 (i.e., $t(A) = t(B) = 9$), though this is not pictured. The duration $\mathcal{D}(1 \to 2)$ of the edge $1 \to 2$ is 6; this is pictured adjacent to the edge $1 \to 2$. After this duration has passed, the manager

Though condition (1.0.1) holds, still there is no solution to the given scheduling problem: A requires four months of time during the first six months, and so does B, but only six months are available.

Run ρ in R	$\sigma(\rho, A)$	$\sigma(\rho, B)$
$(1, 2)$	3	3
$(1, 2, 3)$	6	0
$(1, 2, 4)$	0	6

Figure 2.0.2: A strategy for the problem of Figure 2.0.1.

decides which device to build. To build device A, the manager follows the edge $2 \to 3$. After an additional 6 time units (the duration $\mathcal{D}(2 \to 3)$ is 6), job A is due at vertex 3. The jobs due at vertex 3 are indicated with the label $d : A$ adjacent to vertex 3. To build device B instead, the manager follows the edge $2 \to 4$; after an additional 6 time units, job B is due at vertex 4. □

We will be concerned with a game played by the scheduler versus the environment. The environment decides what branches to take in the conditional graph, and the scheduler decides how to allocate a single processor among the jobs J. At time 0, the game is at the initial vertex v_0. The environment chooses any vertex v_1 such that $v_0 \to v_1$. The scheduler is informed immediately, at time 0, of the environment's decision. During the next $\mathcal{D}(v_0 \to v_1)$ time units, the scheduler allocates the processor among the jobs. In this section, we let scheduler make preemptions at arbitrary times.[2] Each release of a job j is a request for the execution of an additional instance of j. When a vertex v is encountered such that $j \in d(v)$, all previously released instances of j must be complete. Thus, the scheduler loses if some job in both $r(v_0)$ and $d(v_1)$ is not complete by time $\mathcal{D}(v_0 \to v_1)$. Otherwise, the game continues, and the environment again chooses any vertex v_2 such that $v_1 \to v_2$. The scheduler, informed of this choice, allocates the processor among jobs for the next $\mathcal{D}(v_1 \to v_2)$ time units. The scheduler loses if some job in both $d(v_2)$ and $(r(v_0) \setminus d(v_1)) \cup r(v_2)$ is not complete at time $\mathcal{D}(v_0 \to v_1) + \mathcal{D}(v_1 \to v_2)$. The game continues in this way forever, or until a vertex is entered that has no outgoing edges.

Our main goal in this section is to develop an algorithm for finding winning strategies for the scheduler. To this end, we make precise the game we have informally described by defining objects that capture the moves of environment (runs), and the decisions of the scheduler (strategies).

Definition 2.0.3 (runs and strategies). Let P be a conditional scheduling problem. A *run* ρ of P is a sequence $(v_0, v_1, \cdots, v_n)$ of vertices such that $n \geq 1$ and $(v_i, v_{i+1}) \in \mathcal{E}$ for $i \in [0 .. n-1]$. The *length* of run $(v_0, v_1, \cdots, v_n)$ is $n+1$. Let R be the set of runs of P. For any run $\rho = (v_0, \cdots, v_n)$, $(v_0, \cdots, v_n, \cdots, v_m) \in R$ is a *continuation* of ρ, and is a *maximal* continuation if vertex v_m has no successors. A *strategy* σ for P is a function $\sigma : R \times J \to \mathbb{R}^{\geq 0}$ such that for any run $\rho = (v_0, v_1, \cdots, v_n)$ in R,

$$\sum_{j \in J} \sigma(\rho, j) \leq \mathcal{D}(v_{n-1} \to v_n) \tag{2.0.2}$$

If $\sigma(\rho, j) > 0$, we say that strategy *executes* job j along run ρ. For each integer $i \in [0 .. n]$, let $\tau(i) = \sum_{k=0}^{i-1} \mathcal{D}(v_k \to v_{k+1})$. We say that $\tau(i)$ is the time at which the i-th element v_i of run ρ is *entered.* □

Note that $\tau(v_i) - \tau(v_{i-1}) = \mathcal{D}(v_{i-1} \to v_i)$. Intuitively, a strategy σ allocates $\sigma(\rho, j)$ time to job j between times $\tau(n-1)$ and $\tau(n)$. The inequalities (2.0.2) express the constraint that for a run $(v_0, \cdots, v_n)$, the strategy σ allocates at most $\mathcal{D}(v_{n-1} \to v_n)$ from time $\tau(v_{n-1})$ until time $\tau(v_n)$. Note that a strategy is *non-clairvoyant* in the sense that, past the next vertex v_n chosen by the environment, the scheduler has no knowledge of the future behavior of the environment.

Example (2.0.2 continued). For the problem of Figure 2.0.1, the set R of runs is $\{(1, 2), (1, 2, 3), (1, 2, 4)\}$. The strategy presented at the beginning of the report — divide the first six months between A and B, and spend the next six months exclusively on either A or B — is presented in Figure 2.0.2. □

[2] In Sections ?? and ??, we will investigate variants of the conditional scheduling problem in which the scheduler may preempt only at integral times. This models a periodic timer interrupt, for example. The choice of the integers over, say, a set of evenly spaced rationals is arbitrary, but is no less general.

We now define the conditions under which a strategy σ is *winning*. Informally, the definition is as follows. Consider a run $\rho = (v_0, v_1, \cdots, v_n)$, and a job j that is released at v_i for some $i \in [0 \mathrel{..} n-1]$. If there is no vertex v_k, $k \in [i+1 \mathrel{..} n]$, at which j is due, then ρ imposes no requirements on σ. Now suppose there is such a vertex, and let v_{k^*} be such a vertex with minimum index k^*. Each vertex at or after v_i and before v_{k^*} that releases j incurs a requirement of $t(j)$ additional time units for j. Let m denote the number of such vertices. In order to be winning, σ must allocate at least $m \cdot t(j)$ time units for j from time $\tau(i)$ up to time $\tau(k^*)$. More precisely, we define a winning strategy as follows.

Definition 2.0.4. Let P be a conditional scheduling problem. A strategy is *winning* for P if, for every run $\rho = (v_0, v_1, \cdots, v_n) \in R$, for each integer $i \in [0 \mathrel{..} n-1]$, and for each job $j \in r(v_i)$, either the set

$$\{k \mid k \in [i+1 \mathrel{..} n] \text{ and } j \in d(v_k)\} \tag{2.0.3}$$

is empty, or the condition, marked (2.0.4) below, holds. Let k^* be the minimum over the set (2.0.3). Let m be the size of the set $\{\ell \mid i \leq \ell < k^* \text{ and } j \in r(v_\ell)\}$. Then

$$\sum_{\ell=i+1}^{k^*} \sigma((v_0, \cdots, v_\ell), j) \geq m \cdot t(j) \tag{2.0.4}$$

must hold. □

We now consider how to find a winning strategy for a conditional scheduling problem P. On the positive side, if the graph $(\mathcal{V}, \mathcal{E})$ is a tree rooted at v_0, we develop in Section 2.1 a polynomial-time algorithm that determines whether P has a winning strategy, and if so synthesizes such a strategy. We then show how to enrich the tree scheduling model with an anytime reward function (Section 2.2) and precedence constraints (Section 2.3) while retaining a polynomial-time synthesis algorithm. On the negative side, if $(\mathcal{V}, \mathcal{E})$ is a directed acyclic graph (respectively, if strategies must be discrete-time), we show in Section 3 that determining whether P has a feasible strategy is coNP-hard (respectively, NP-hard). Table 5.0.5 on page 29 summarizes the results of this report.

2.1 Tree scheduling

We now present an algorithm that decides whether P has a winning strategy, and if so returns such a strategy. This algorithm (1) creates a system of linear inequalities that captures the constraints on a winning strategy, and (2) tests these inequalities for a solution using a polynomial-time linear programming algorithm. The inequalities have the property that any solution corresponds to a winning strategy. Further, the inequalities may be generated in time, and *a fortiori* size, that are polynomial in the conditional scheduling problem, *if* the graph $(\mathcal{V}, \mathcal{E})$ is a tree rooted at v_0. Thus, for such tree-shaped problems we will show that our algorithm runs in polynomial time.[3] It should be emphasized that if $(\mathcal{V}, \mathcal{E})$ is instead a directed acyclic graph, the running time may not be bounded by a polynomial, though our algorithm still synthesizes a winning strategy if one exists.

In order to present our algorithm, we introduce the system of linear inequalities generated in step (1) of our algorithm by means of an example.

Example (2.0.2 continued). For our running example, the inequalities are

$$\begin{array}{rclrcl} \sigma((1,2),A) & \geq & 0 & \sigma((1,2),B) & \geq & 0 \\ \sigma((1,2,3),A) & \geq & 0 & \sigma((1,2,3),B) & \geq & 0 \\ \sigma((1,2,4),A) & \geq & 0 & \sigma((1,2,4),B) & \geq & 0 \end{array} \tag{2.1.1}$$

$$\begin{array}{rcl} \sigma((1,2),A) + \sigma((1,2),B) & \leq & 6 \\ \sigma((1,2,3),A) + \sigma((1,2,3),B) & \leq & 6 \\ \sigma((1,2,4),A) + \sigma((1,2,4),B) & \leq & 6 \end{array} \tag{2.1.2}$$

[3] Of course, a nonpolynomial-time algorithm, such as the simplex method, may in practice run more quickly. Our focus here is not to find the fastest algorithm in practice, but instead to prove the existence of a polynomial-time algorithm.

$$\begin{array}{rcl} \sigma((1,2),A)+\sigma((1,2,3),A) & \geq & 9 \\ \sigma((1,2),B)+\sigma((1,2,4),B) & \geq & 9 \end{array} \tag{2.1.3}$$

The variables of the inequalities are the members of the set $\{\sigma(\rho,j) \mid \rho \in R \wedge j \in J\}$. The inequalities (2.1.1) require that $\sigma(\rho,j)$ is nonnegative for each run $\rho \in R$ and job $j \in J$. The inequalities (2.1.2) capture the constraint (2.0.2), that the amount of time allocated by the scheduler during an interval is at most the duration of that interval. Any assignment of values to the variables $\sigma(\rho,j)$ that satisfies (2.1.1) and (2.1.2) also satisfies the requirements of Definition 2.0.3; such an assignment is thus a strategy. The inequalities (2.1.3) express the constraint (2.0.4), that between the release of a job and its next subsequent deadline, sufficient time is allocated to that job. The reader may verify that the strategy of Figure 2.0.2 satisfies the inequalities (2.1.1), (2.1.2), and (2.1.3). □

We now formally define the system $Lin[P]$ of linear inequalities generated by a conditional scheduling problem P. There will be finitely many inequalities if the graph $(\mathcal{V},\mathcal{E})$ is a directed acyclic graph, and will be polynomial in the size of P if $(\mathcal{V},\mathcal{E})$ is a tree rooted at v_0.

Definition 2.1.1 (the system $\boldsymbol{Lin[P]}$ of linear inequalities). Let P be a conditional scheduling problem. The set of variables of $Lin[P]$ is $\{\sigma(\rho,j) \mid \rho \in R \wedge j \in J\}$. There are three types of constraints in $Lin[P]$:

- *[Nonnegativity constraints]* For each variable $\sigma(\rho,j)$, $\sigma(\rho,j) \geq 0$ is a constraint.
- *[Duration constraints]* For each run $\rho = (v_0,\cdots,v_n) \in R$, $\sum_{j\in J}\sigma(\rho,j) \leq \mathcal{D}(v_{n-1} \to v_n)$ is a constraint.
- *[Execution time constraints]* For each run $\rho = (v_0,\cdots,v_n) \in R$, for each integer $i \in [0 \mathinner{..} n-1]$, for each job $j \in r(v_i)$, if the set (2.0.3) is nonempty, then $\sum_{\ell=i+1}^{k^*}\sigma((v_0,\cdots,v_\ell),j) \geq m \cdot t(j)$ is a constraint, where k^* and m are as defined in Definition 2.0.4. □

The reader will note the close correspondence between Definition 2.0.3 and the nonnegativity and interval constraints of $Lin[P]$. Based on this correspondence, it is straightforward to prove the following proposition.

Proposition 2.1.2. An assignment of values to the variables of $Lin[P]$ is a strategy if and only if the assignment satisfies the nonnegativity constraints and the interval constraints.

The reader will also note the close correspondence between Definition 2.0.4 and the execution time constraints of $Lin[P]$. It is thus straightforward to prove that an assignment of values to the variables of $Lin[P]$ is a winning strategy if and only if the assignment is a strategy and moreover satisfies the execution time constraints of $Lin[P]$. From Proposition 2.1.2, the following proposition follows.

Proposition 2.1.3. An assignment of values to the variables of $Lin[P]$ is a winning strategy if and only if the assignment satisfies the nonnegativity, duration, and execution time constraints.

We now bound the running time of our algorithm. In order to do so, we bound the size of $Lin[P]$. Since the graph $(\mathcal{V},\mathcal{E})$ is a tree rooted at v_0, the number $|R|$ of runs equals the number $|\mathcal{E}|$ of edges. The number of variables of $Lin[P]$ equals $|R| \cdot |J| = |\mathcal{E}| \cdot |J|$. Since there is one nonnegativity constraint per variable, the number of these constraints is also $|\mathcal{E}| \cdot |J|$. Since there is one duration constraint per run, the number of these constraints is $|\mathcal{E}|$. Each duration constraint is the sum of at most $|J|$ terms, for a total size of $O(|\mathcal{E}| \cdot |J|)$. The number of execution time constraints is at most the number of runs, times the length of the longest run, times the number of jobs; and the number of summands in each execution time constraint is at most the length of the longest run. The longest run has length at most $|\mathcal{E}|$, for a total of size $O(|\mathcal{E}|^3 \cdot |J|)$. We conclude that the size of $Lin[P]$ is polynomial in the size of P. Further, it is straightforward to verify that $Lin[P]$ may be generated in time polynomial in the size of P. Since a polynomial-time linear programming algorithm may be used to test whether $Lin[P]$ has a solution, we have established the following theorem.

Theorem 2.1.4. Let P be a conditional scheduling problem in which $(\mathcal{V},\mathcal{E})$ is a tree rooted at v_0. Then the algorithm of this section is polynomial time, determines whether a winning strategy for P exists, and if so returns such a strategy.

2.2 Imprecise tree scheduling

It has been widely observed within the artificial intelligence community that the amount of time required to compute an optimal result may reduce the utility of the result [RW91]. Since earlier results are generally better than later results in a real-time setting, computing the optimal result after a long delay may be less desirable than computing a sub-optimal result after a short delay. In order to maximize a utility function, decisions need to be made about the amount of time to devote to each job. Anytime algorithms allow such decisions to be made flexibly [DB88]. An *anytime algorithm* is an algorithm in which computation may be interrupted at any time, producing results of increasing quality as the amount of computation time increases.

The similar concept of *imprecise computations* has been studied in scheduling theory since the 1980s [SLC91, LLSY91]. An imprecise computation consists of two parts, a *mandatory* part and an *optional* part. The mandatory part must be completed to produce a result of minimum acceptable quality. The optional part follows the mandatory part, and improves the result produced by the mandatory part. [SLC91] presents a polynomial-time algorithm that finds, from among all schedules satisfying release, deadline, and mandatory computation constraints, one that is optimal in the sense of minimizing the total amount of remaining optional computation. In addition, [SLC91] permits a positive, rational weight for each job, and shows how to minimize the weighted sum of remaining computation times. The per-job weight may be thought of as a linear *reward function*. [AMMMA01] generalizes [SLC91] to include concave reward functions, and presents a polynomial-time algorithm that finds optimal schedules for this more general model, as long as strong periodicity requirements are met.

The conditional scheduling problem of Definition 2.0.1 can easily be adapted to fit the framework of imprecise computations. The condition (2.0.4) specifies a lower bound on the amount of time each job j must be executed — or, in other words, the mandatory execution time of j. Any additional execution time is optional. To quantify the total reward of a strategy, we augment the basic conditional scheduling problem with a reward function $f : J \to \mathbb{Q}^{\geq 0}$. As we will see, this reward function behaves similarly to the linear reward function of [SLC91].

Definition 2.2.1 (imprecise scheduling problem). An *imprecise scheduling problem* P is a triple $(\mathcal{F}, W, f)$, where the finite state machine $\mathcal{F}$ and workload W are defined as in Definition 2.0.1, and $f : J \to \mathbb{Q}^{\geq 0}$ is a function, called the *reward function*, assigning a non-negative rational number $f(j)$ to each job j. The *runs*, *strategies*, and *winning* strategies of an imprecise scheduling problem $(\mathcal{F}, W, f)$ are the same as for the underlying conditional scheduling problem $(\mathcal{F}, W)$. □

Our goal is to develop an algorithm for finding a winning strategy of maximum reward. Before we define what the reward of a strategy is, however, we restrict the class of imprecise scheduling problems that we will consider in four ways. The intent of these restrictions is (1) to simplify the definition of the reward of a strategy, and (2) to focus on a class of imprecise scheduling problems for which a polynomial-time algorithm exists. To these ends, we define a *well-formed* imprecise scheduling problem as follows:

Definition 2.2.2 (well-formed imprecise scheduling problem). We say that an imprecise scheduling problem is *well-formed* if the following conditions hold:

1. In order to develop a polynomial-time algorithm, we require that the graph $(\mathcal{V}, \mathcal{E})$ is a tree rooted at v_0.

2. In order to simplify the definition of the reward of a strategy, we require that for each job $j \in J$, there exists exactly one vertex $v \in \mathcal{V}$ such that $j \in r(v)$. Given this requirement, for $j \in J$ we let $r(j)$ denote the unique vertex $v \in \mathcal{V}$ such that $j \in r(v)$.

3. Without loss of generality, we require that for any $v \in \mathcal{V}$ and any $j \in d(v)$, there exists $v' \in \mathcal{V}$ such that $v' \to^+ v$ and $j \in r(v')$.[4]

[4] No generality is lost for the following reason. If $j \in d(v)$, but there does not exist a $v' \in \mathcal{V}$ with $v' \to^+ v$ and $j \in r(v')$, then upon reaching v, j cannot have been released. Thus, the fact that j is due at v can be ignored. A problem not satisfying 3 may be replaced by a problem that does satisfy 3, by removing j from $d(v)$; the constraints on a winning strategy remain the same.

4. Given 1 and 2, without loss of generality we require that there do not exist two vertices $v, v' \in \mathcal{V}$ such that $v \rightarrow^+ v'$ and $d(v) \cap d(v') \neq \emptyset$.[5] □

We now define the reward of a strategy. The reward of a run $(v_0, \cdots, v_n)$ is the sum, over all jobs j due at v_n, of $f(j)$ times the amount of time allocated to j since the vertex releasing j was entered.[6] The reward of a strategy is the sum, over all runs $\rho \in R$, of the reward of ρ. We formalize this notion in the following definition:

Definition 2.2.3 (reward of a strategy). Let σ be a strategy for a well-formed imprecise scheduling problem. The *reward* of σ is

$$\sum_{(v_0,\cdots,v_n)\in R} \; \sum_{j \in d(v_n)} f(j) \cdot \sum_{i=r(j)+1}^{n} \sigma((v_0, \cdots, v_i), j) \tag{2.2.1}$$

We use the symbol $f[\sigma]$ to denote the reward of strategy σ.[7] We say that a strategy σ is *optimal* if σ is winning, and for all winning strategies σ', $f[\sigma] \geq f[\sigma']$. □

The third (last) summation in (2.2.1) measures the amount of time allocated to job j between entering the vertex $v_{r(j)}$ that releases j and entering the vertex v_n at which j is due. The second summation measures the reward of the run $(v_0, \cdots, v_n)$. The first summation, of course, measures the reward of the strategy σ.

We wish to develop an algorithm that, when given a well-formed imprecise scheduling problem $P = (\mathcal{F}, W, f)$, decides whether P has a winning strategy, and if so returns an optimal strategy. Given our linear programming approach, this is quite easily accomplished: we simply use a polynomial-time linear programming algorithm to maximize the objective function (2.2.1) subject to the constraints $Lin[P']$, where $P' = (\mathcal{F}, W)$. We have thus established the following theorem:

Theorem 2.2.4. Let P be a well-formed imprecise scheduling problem. Then the algorithm of this section is polynomial time, decides whether P has a winning strategy, and if so returns an optimal strategy.

2.3 Precedence-constrained tree scheduling

In this section, we enrich the conditional scheduling model of Definition 2.0.1 by adding precedence constraints. We then develop a polynomial-time algorithm, based on the algorithm of Section 2.1, to synthesize schedules for the extended model. We begin by adding a precedence relation $\prec \subset J \times J$ to the conditional scheduling problem:

Definition 2.3.1 (precedence-constrained scheduling problem). A *precedence-constrained scheduling problem* P is a triple $(\mathcal{F}, W, \prec)$, where the finite state machine $\mathcal{F}$ and workload W are defined as in Definition 2.0.1, and the *precedence relation* $\prec \subset J \times J$ is an acyclic binary relation on J. We shall normally write $j \prec j'$ instead of $(j, j') \in \prec$. The *runs* and *strategies* of a precedence-constrained scheduling problem $(\mathcal{F}, W, \prec)$ are the same as for the underlying conditional scheduling problem $(\mathcal{F}, W)$. □

We will consider a restricted subset of precedence-constrained scheduling problems. As in Definition 2.2.2, we require that the graph $(\mathcal{V}, \mathcal{E})$ is a tree rooted at v_0, and that each job is released by exactly one vertex. The first requirement is necessary for a polynomial-time strategy synthesis algorithm. The second requirement, in the present context, simplifies the definition of a winning strategy (Definition 2.3.3, below). Without loss of

[5] No generality is lost for the following reason. Given 2, for any run $(v_0, \cdots, v_n)$ and any job j, there is at most one vertex v_i such that $j \in r(v_i)$. Now if j were in both $d(v_k)$ and $d(v_{k'})$, for some $i + 1 \leq k < k' \leq n$, then given 1, $v_{k'}$ is only reachable from v_i by first passing through v_k. Thus, the fact that j is due at $v_{k'}$ can be ignored, since j already had to complete before the predecessor v_k of $v_{k'}$ is reached. Any problem satisfying 1 and 2 but not 4 can be replaced by one that satisfies 1, 2, and 4, by removing j from $d(v_{k'})$; the constraints on a winning strategy remain the same.

[6] Given the definition of well-formedness, j was released by some previous vertex v_i, $i \in [0 \,..\, n-1]$; j was released by exactly one such vertex; and j was not due at some previous vertex v_k, $i \in [i+1 \,..\, n-1]$.

[7] Other definitions of the reward of a strategy are also possible: one might, for example, weight the last sum by $f(j)/n(j)$, where $n(j)$ is the number of vertices at which j is due. Alternatively, one might weight the second sum by $Prob(\rho)$, where $Prob$ is a probability distribution over runs. The significant choice is not the specific way in which the reward of a strategy is defined, but rather that the definition be a linear function of the variables $\sigma((v_0, \cdots, v_i), j)$.

generality, we also require that conditions 3 and 4 of Definition 2.2.2 hold. Finally, we require that if $j \prec j'$, then j is both released and due prior to j'. We now precisely specify the class of precedence-constrained scheduling problems we will consider:

Definition 2.3.2 (well-formed precedence-constrained scheduling problem). We say that a precedence-constrained scheduling problem is *well-formed* if:

1. Conditions 1, 2, 3, and 4 of Definition 2.2.2 hold.
2. For any two jobs $j, j' \in J$, if $j \prec j'$, then $r(j) \to^* r(j')$.
3. Let j, j' be any two members of J such that $j \prec j'$. Suppose $j' \in d(v')$ for some $v' \in \mathcal{V}$. Then there exists $v \in \mathcal{V}$ such that $j \in d(v)$ and $v \to^* v'$. □

We now consider what it means for a strategy to be winning for a well-formed precedence-constrained scheduling problem. Consider a run $(v_0, \cdots, v_n) \in R$, and an $i \in [0 \mathrel{..} n-1]$. Since $\prec$ is acyclic, for any strategy σ, it is possible to topologically sort the the execution of jobs from time $\tau(i)$ until time $\tau(i+1)$, so that $\prec$ is respected. On the other hand, suppose that for two jobs $j, j' \in J$ such that $j \prec j'$, there exists a $k \in [i+1 \mathrel{..} n]$ such that

$$\begin{aligned} \sigma((v_0, \cdots, v_i), j') &> 0 \\ \sigma((v_0, \cdots, v_k), j) &> 0 \end{aligned}$$

In this case, strategy σ violates the precedence constraint $j \prec j'$, since j does not finish before j' begins. If no such violation occurs, σ is winning. More precisely, we define a winning strategy as follows:

Definition 2.3.3 (winning strategy). Let $P = (\mathcal{F}, W, \prec)$ be a well-formed precedence-constrained scheduling problem. A strategy σ is *winning* for P if σ is winning for $(\mathcal{F}, W)$ in the sense of Definition 2.0.4, and the following additional condition holds. Consider any run $(v_0, \cdots, v_n) \in R$ and any jobs $j, j' \in J$ such that $j \prec j'$. If $\sigma((v_0, \cdots, v_i), j') > 0$ for some $i \in [1 \mathrel{..} n-1]$, then $\sigma((v_0, \cdots, v_k), j) = 0$ for all $k \in [i+1 \mathrel{..} n]$. In other words,

$$\begin{aligned} &\forall i \in [1 \mathrel{..} n-1] \\ &\quad \forall k \in [i+1 \mathrel{..} n] \\ &\qquad \sigma((v_0, \cdots, v_i), j') > 0 \quad \Longrightarrow \quad \sigma((v_0, \cdots, v_k), j) = 0 \end{aligned} \tag{2.3.1}$$

If (2.3.1) does not hold for some run $\rho = (v_0, \cdots, v_n)$, some precedence constraint $j \prec j'$, and some $i \in [1 \mathrel{..} n-1]$, then we say that σ *i-violates* precedence constraint $j \prec j'$ along run ρ. □

Suppose that $(\mathcal{F}, W, \prec)$ is a precedence-constrained scheduling problem, and that σ is a winning strategy for the less constrained conditional scheduling problem $(\mathcal{F}, W)$. Somewhat surprisingly, a strategy σ' that is winning for $(\mathcal{F}, W, \prec)$ may be derived from σ in polynomial time. The relation between precedence constrained scheduling and conditional scheduling is analogous to the relation between the classical scheduling problems $1 \mid r_j;\, d_j;\, prec;\, pmtn \mid -$ and $1 \mid r_j;\, d_j;\, pmtn \mid -$ (see [Bła76]). In each case, schedules for the precedence-constrained version can be derived from schedules for the version without precedence constraints by an appropriate topological sort. In the present case, the topological sort required is more complex, but the basic intuition remains the same: if job j precedes job j', then it is always acceptable to execute j in preference to j'. We now show how to derive a winning strategy for $(\mathcal{F}, W, \prec)$ from a winning strategy for $(\mathcal{F}, W)$ in polynomial time.

Proposition 2.3.4. Let $P = (\mathcal{F}, W, \prec)$ be a well-formed precedence-constrained scheduling problem. Let σ be a winning strategy for $(\mathcal{F}, W)$ in the sense of Definition 2.0.4. Then a winning strategy for P may be obtained from σ in time polynomial in $|\mathcal{V}| \cdot |J|$.

Proof. Without loss of generality, we assume that the winning strategy σ for $(\mathcal{F}, W)$ has three additional properties. Let $\rho = (v_0, \cdots v_n) \in R$ be any run, and let $j \in J$ be any job. Then:

I. If $\sigma(\rho, j) > 0$ then there exists a vertex $v \in \mathcal{V}$ such that $j \in d(v)$ and $v_n \to^* v$. Informally, j is not executed along run ρ unless it is subsequently due along a continuation of ρ.

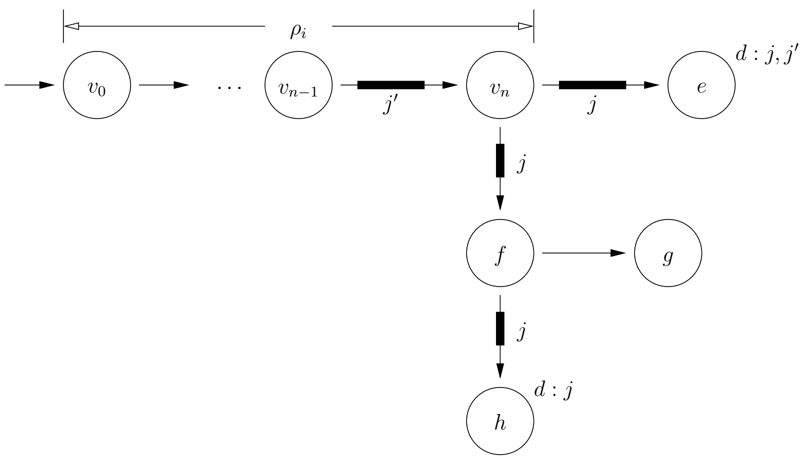

Figure 2.3.1: A visual aid for the proof of Lemma 2.3.5.

II. If $\sigma(\rho, j) > 0$ then $j \in r(v_i)$ for some $i \in [0 .. n-1]$. Informally, j is not executed along run ρ unless it has been released.

III. $\sum_{i=1}^{n} \sigma((v_0, \cdots, v_i), j) \leq t(j)$. Informally, j is not executed for more than $t(j)$ time units.

Let runs $\rho_1, \cdots, \rho_m$ be the members of R, enumerated in order of nondecreasing length. Let $\sigma_0 = \sigma$. For $i = 1, \cdots, m$, we construct a strategy σ_i from σ_{i-1}, with properties I–III above, and two additional properties:

IV. σ_i remains winning for $(\mathcal{F}, W)$.

V. Let n be the length of ρ_i, and let ρ be any continuation of a run in $\{\rho_1, \cdots, \rho_i\}$. Then for any $k \in [1 .. n-1]$, there is no precedence constraint $j \prec j'$ such that σ_i k-violates $j \prec j'$ along ρ.

Properties IV and V imply that σ_m is winning for $(\mathcal{F}, W, \prec)$. Since $(\mathcal{V}, \mathcal{E})$ is a tree, the number m of runs is $O(|\mathcal{V}|)$, and the following lemma completes the proof. □

Lemma 2.3.5. Strategy σ_i may be obtained from strategy σ_{i-1} in time polynomial in $|\mathcal{V}| \cdot |J|$.

Proof. Let $(v_0, \cdots, v_n)$ be the sequence of vertices of run ρ_i. Let $j, j' \in J$ be any two jobs such that $j \prec j'$ and σ_{i-1} n-violates $j \prec j'$ along some continuation of ρ_i. Consider any maximal continuation $\rho = (v_0, \cdots, v_n, \cdots, v_{n'})$ of ρ_i such that $j \in d(v_k)$ for some $k \in [n+1 .. n']$. We call such a continuation a (ρ_i, j)-*continuation.* Since σ_{i-1} is winning for $(\mathcal{F}, W)$ and satisfies III,

$$\sum_{\ell=n+1}^{n'} \sigma_{i-1}((v_0, \cdots, v_\ell), j) \quad = \quad t(j) - \sum_{\ell=1}^{n} \sigma_{i-1}((v_0, \cdots, v_\ell), j)$$

In other words, for any (ρ_i, j)-continuation, the amount of time allocated to j after vertex v_n is the same, and is equal to the time $t(j)$ required by j minus the amount of time allocated to j before v_n. Figure 2.3.1 illustrates this situation. In the figure, after v_n, the same amount of time is allocated to j on each (ρ_i, j)-continuation (these continuations are $(v_0, \cdots, e)$ and $(v_0, \cdots, h)$). On other continuations, less time is allocated to j (this other continuation is $(v_0, \cdots, g)$).

Select two jobs j, j' such that (1) $j \prec j'$, (2) σ_{i-1} n-violates $j \prec j'$ along some continuation of ρ_i, and (3) j is minimal in the partial order $\prec^+$. Let T be the minimum of the amount $\sigma_{i-1}(\rho_i, j')$ of time allocated to j' along run ρ_i, and the amount that remains to be executed of job j, i.e.,

$$T = \min\{\sigma_{i-1}(\rho_i, j'), \quad t(j) - \textstyle\sum_{i=1}^{n} \sigma_{i-1}((v_0, \cdots, v_i), j)\}$$

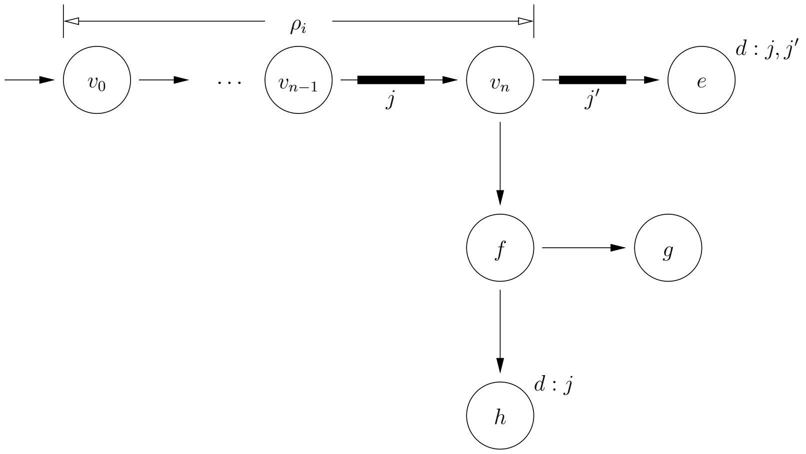

Figure 2.3.2: The transformation of the strategy from Figure 2.3.1.

We now perform an exchange of execution times. We move T units of the execution of j from the (ρ_i, j)-continuations to ρ_i, so that $\sigma_i(\rho_i, j) = T$. At the same time, we move T units of the execution of j' from ρ_i to the times just vacated in the (ρ_i, j)-continuations, taking care to preserve property I by not executing j' along continuations on which j' is not due. This transformation is illustrated in Figure 2.3.2, where σ_{i-1} was the strategy of Figure 2.3.1, and σ_{i-1} is the strategy of Figure 2.3.2. Note in Figure 2.3.2 that j' is executed along $(v_0, \cdots, e)$, since $j' \in d(e)$; but that to preserve property I, j' is not executed along either $(v_0, \cdots, f)$, $(v_0, \cdots, g)$, or $(v_0, \cdots, h)$, since j' is not in either $d(f)$ or $d(h)$.

We repeat this process with additional jobs j, j' satisfying (1)–(3) until no such jobs remain, thus obtaining strategy σ_i. There are at most $\binom{|J|}{2}$ exchanges to perform. Each exchange takes $O(|\mathcal{V}|)$ time, since there are $O(|\mathcal{V}|)$ continuations of ρ_i. Thus, obtaining σ_i requires time polynomial in $|\mathcal{V}| \cdot |J|$. We now prove that properties I–V hold for σ_i.

I. Consider any jobs j, j' that were exchanged in the construction of σ_i. Since σ_{i-1} satisfies property I, and j is executed by σ_{i-1} along a continuation of ρ_i, j is due along a continuation of ρ_i. Similarly, j' is due along a continuation of ρ_i, since σ_{i-1} satisfies I, and j is executed by σ_{i-1} along ρ_i. Further, by the construction of σ_i, j' is not executed along those continuations on which j' is not due. Thus, I holds for σ_i.

II. Consider again any jobs j, j' that were exchanged. Since σ_{i-1} satisfies II, and $\sigma_{i-1}(\rho_i, j') > 0$, $j' \in r(v_k)$ for some $k \in [0 \mathinner{..} n-1]$. By condition 2 of Definition 2.3.2, $j \in r(v_{k'})$ for some $k' \in [0 \mathinner{..} k]$. Thus, II holds for σ_i.

III. Consider again any jobs j, j' that were exchanged. Then for each leaf vertex $v_\ell \in \mathcal{V}$,

$$\textstyle\sum_{k=1}^{\ell} \sigma_i((v_0, \cdots, v_k), j) = \sum_{k=1}^{\ell} \sigma_{i-1}((v_0, \cdots, v_k), j)$$

whereas

$$\textstyle\sum_{k=1}^{\ell} \sigma_i((v_0, \cdots, v_k), j') \le \sum_{k=1}^{\ell} \sigma_{i-1}((v_0, \cdots, v_k), j')$$

Since σ_{i-1} satisfied III, σ_i satisfies III as well.

IV. Consider again any jobs j, j' that were exchanged. Job j is not moved earlier than the vertex that releases it, by II. Since by the argument for III, the total time allocated to j remains the same, the inequality (2.0.4) continues to hold for job j. For job j', by property I for σ_{i-1}, the times vacated by j do not occur past the vertices at which j is due. By condition 3 of Definition 2.3.2, *a fortiori* the times vacated by j do not occur past any vertex at which j' is due. Thus, j' is not moved later than any vertex at which it is due. Further, along the paths on which j' is due, the time allocated to j' remains the same. Thus, (2.0.4) continues to hold for job j'. We conclude that σ_i remains feasible for $(\mathcal{F}, W)$.

V. Each n-violation of a precedence constraint by σ_{i-1} was removed in the construction of σ_i. Further, for $k < n$, no k-violations were reintroduced in the construction of σ_i, since the construction of σ_i only modified the behavior of σ_{i-1} for ρ_i and continuations of ρ_i. Thus, σ_i satisfies V. □

Consider a well-formed precedence-constrained scheduling problem $(\mathcal{F}, W, \prec)$. If $(\mathcal{F}, W)$ has a winning strategy σ, then a winning strategy for $(\mathcal{F}, W, \prec)$ may be constructed from σ in polynomial time. On the other hand, if $(\mathcal{F}, W)$ has no winning strategy, then $(\mathcal{F}, W, \prec)$ has no winning strategy either, since $(\mathcal{F}, W)$ is a less constrained version of $(\mathcal{F}, W, \prec)$. Thus, to check whether $(\mathcal{F}, W, \prec)$ has a winning strategy, the following algorithm suffices: first, test whether $Lin[\mathcal{F}, W]$ has a feasible solution σ. If not, then report that $(\mathcal{F}, W, \prec)$ has no winning strategy. If so, use the algorithm of Proposition 2.3.4 to obtain a winning strategy from the feasible solution σ. We have established the following theorem:

Theorem 2.3.6. Let $P = (\mathcal{F}, W, \prec)$ be a well-formed precedence-constrained scheduling problem. Then the algorithm of this section is polynomial time, determines whether a winning strategy for P exists, and if so returns such a strategy.

3 Hard conditional scheduling problems

In Section 2, we saw that several conditional scheduling problems — tree scheduling, imprecise tree scheduling, and precedence-constrained tree scheduling — can be solved in polynomial time. In this section, we will investigate conditional scheduling problems that cannot be solved in polynomial time, unless P = NP. Section 3.1 examines *discrete-time* strategies, in which the scheduler is restricted to make decisions only at integral points in time. We will see that determining whether a tree scheduling problem has a winning discrete-time strategy is NP-hard. Section 3.2 considers conditional scheduling problems in which the graph $(\mathcal{V}, \mathcal{E})$ is a directed acyclic graph (DAG). We will see that determining whether a DAG scheduling problem has a winning strategy is coNP-hard. Finally, Section 3.3 shows that determining whether a DAG scheduling problem has a winning discrete-time strategy is PSPACE-hard.

3.1 Discrete-time tree scheduling

There is one respect in which the strategies produced by the algorithm of Section 2.1 are impractical: they require that preemptions be made at arbitrary points in time. This is not possible in computer systems, since the clock rate provides an upper bound on the frequency of preemptions; moreover, because of context-switch overhead, most real-time systems do not function well with timers that run faster than 10 kHz. Suppose that strategies are restricted to preempt only at a sparse, evenly-spaced spaced set of times — at integral times, say.[8] We call such restricted strategies *discrete-time* strategies. For discrete-time strategies, do tree scheduling problems remain solvable in polynomial time? Unfortunately, the answer is negative, as we will see in this section.

The difference in the complexity of dense- and discrete-time versions of tree scheduling provides evidence that conditional scheduling with varying deadlines is fundamentally different from standard single-processor scheduling models. In standard single-processor settings, the same polynomial-time algorithm is often optimal both for a model that allows preemption at any time, and for a model that allows preemption only at integer points in time. In contrast, unless P = NP, there can be no polynomial-time tree scheduling algorithm that is optimal for both dense-time and discrete-time models.

We now precisely define *discrete-time strategies*; we also define the problem *Discrete-time tree scheduling* that we will prove is NP-hard:

Definition 3.1.1 (discrete-time strategy, *Discrete-time tree scheduling*). A strategy σ for a conditional scheduling problem P is *discrete-time* if, for each run $\rho \in R$ and each job $j \in J$, $\sigma(\rho, j) \in \mathbb{Z}$. The

[8] We consider integral times only for reasons of simplicity. The results of this section generalize to any evenly-spaced set of time points.

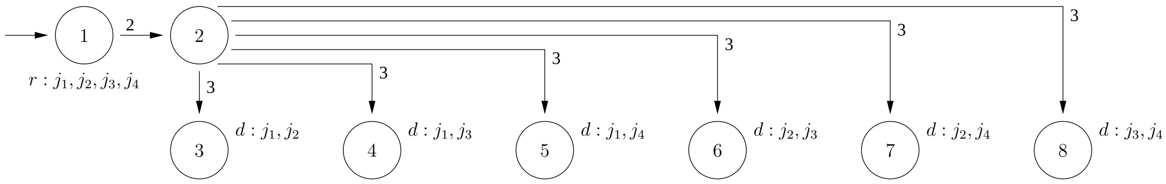

Figure 3.1.1: The conditional scheduling problem of Example 3.1.2.

Run $\rho \in R$	$\sigma(\rho, j_1)$	$\sigma(\rho, j_2)$	$\sigma(\rho, j_3)$	$\sigma(\rho, j_4)$
$(1, 2)$	0.5	0.5	0.5	0.5
$(1, 2, 3)$	1.5	1.5	0	0
$(1, 2, 4)$	1.5	0	1.5	0
$(1, 2, 5)$	1.5	0	0	1.5
$(1, 2, 6)$	0	1.5	1.5	0
$(1, 2, 7)$	0	1.5	0	1.5
$(1, 2, 8)$	0	0	1.5	1.5

Figure 3.1.2: A winning strategy σ for the conditional scheduling problem of Example 3.1.2.

set *Discrete-time tree scheduling* is $\{P \mid P$ is a conditional scheduling problem such that (1) $(\mathcal{V}, \mathcal{E})$ is a tree rooted at v_0, and (2) there exists a winning discrete-time strategy for $P\}$. □

Clearly, since discrete-time strategies are strategies, the existence of a winning discrete-time strategy implies the existence of a winning strategy. However, the converse is not true. As the following example shows, some tree scheduling problems have a winning strategy, but no winning discrete-time strategy.

Example 3.1.2. Consider the tree scheduling problem of Figure 3.1.1. There are four jobs, j_1, j_2, j_3, and j_4. The amount $t(j_i)$ of time required by job j_i is 2 for all $i \in [1 .. 4]$. The initial vertex is 1. At vertex 1, jobs j_1, j_2, j_3, and j_4 are released. Vertex 2 has six successors, one for each of the $\binom{4}{2}$ ways of choosing two of the four jobs. At each successor of vertex 2, two jobs are due, so that every set of two jobs is due at some successor of vertex 2. The duration $\mathcal{D}(1 \to 2)$ of the edge $1 \to 2$ is 2; all other edges have a duration of 3. Figure 3.1.2 depicts the unique winning strategy for this problem. This strategy divides the first 2 time units equally between jobs j_1, j_2, j_3, and j_4, so that $\sigma((1, 2), j) = 0.5$ for each $j \in J$. The remaining 3 units of time are divided equally between the two jobs due at whichever of the successor vertices 3 through 8 is chosen. It may be verified that strategy σ is the only solution to the inequalities $Lin[P]$ of Definition 2.1.1. Note that $\sigma(\rho, j) \notin \mathbb{Z}$ for each run $\rho \in R$ and job $j \in J$. □

In the remainder of this section, we prove the following theorem:

Theorem 3.1.3. *Discrete-time tree scheduling* is NP-hard, even if the time $t(j)$ required by each job j is 1.

We prove this theorem by presenting a polynomial-time reduction from 3-SAT to *Discrete-time tree scheduling*. Recall that a *3-CNF formula*

$$\phi = \bigwedge_{i=1}^{m} \ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3}$$

is a conjunction of *clauses*, of which let us say there are m. Each clause is the disjunction $\ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3}$ of three *literals* $\ell_{i,1}$, $\ell_{i,2}$, and $\ell_{i,3}$. Each literal is either a boolean variable x_k or a negated boolean variable $\neg x_k$. We will let n denote the number of variables appearing in ϕ, and we will assume that these variables are members of the set $\{x_1, \cdots, x_n\}$. Without loss of generality, we may assume that each clause contains three distinct variables [Pap94]. The set *3-SAT* is the set of all 3-CNF formulae that are satisfiable. We now informally describe the reduction from 3-SAT to *Discrete-time tree scheduling*, after which we formally describe the reduction and prove Theorem 3.1.3.

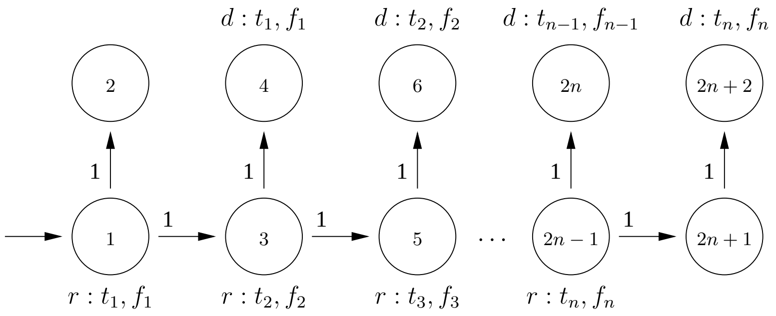

Figure 3.1.3: The assignment gadget.

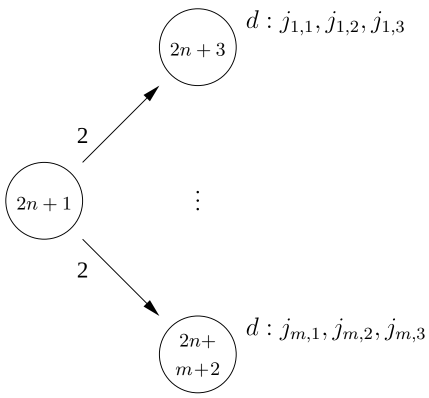

Figure 3.1.4: The formula gadget.

3.1.1 Introduction to the reduction from 3-SAT to *Discrete-time tree scheduling*

Given a 3-CNF formula ϕ, the reduction produces a tree scheduling problem $P[\phi]$, with the following property: $P[\phi]$ has a winning discrete-time strategy if and only if ϕ is satisfiable. The problem $P[\phi]$ consists of two parts, an *assignment gadget* and a *formula gadget*. Figure 3.1.3 shows the assignment gadget. The assignment gadget consists of the $2n+2$ vertices $1, 2, \cdots, 2n+2$. (Vertex 2 plays no significant role in the construction, but is included to make the graph regular.) At vertex $2i-1$, for $i \in [1 \,..\, n]$, jobs t_i and f_i are released. Each job requires one unit of time, i.e., $t(t_i) = t(f_i) = 1$. Jobs t_i and f_i are due at vertex $2i+2$. For any feasible discrete-time strategy σ, either:

$$\begin{array}{rcl} \sigma((1, \cdots, 2i+1), t_i) & = & 1 \quad \text{and} \\ \sigma((1, \cdots, 2i+2), f_i) & = & 1 \end{array}$$

or:

$$\begin{array}{rcl} \sigma((1, \cdots, 2i+1), f_i) & = & 1 \quad \text{and} \\ \sigma((1, \cdots, 2i+2), t_i) & = & 1 \end{array}$$

In an intuitive sense, the first choice corresponds to a truth assignment that gives variable x_i the value *true*, and the second choice corresponds to an assignment that gives x_i the value *false*. Intuitively, then, the assignment gadget forces the scheduling algorithm to pick a truth assignment.

The reduction now needs a means for determining whether the assignment chosen by the scheduling algorithm "satisfies" the 3-CNF formula ϕ. This mechanism is provided by the formula gadget, pictured in Figure 3.1.4. The formula gadget consists of the m vertices $2n+3, \cdots, 2n+m+2$. For $i \in [1 \,..\, m]$, three jobs are due at vertex $2n+i+2$. Which jobs these are depend on the variables occurring in the literals $\ell_{i,1}$, $\ell_{i,2}$, and $\ell_{i,3}$, and whether these literals are positive or negative. To be precise, $d(2n+i+2) = \{j_{i,1}, j_{i,2}, j_{i,3}\}$, where for $i' \in [1 .. 3]$,

$$j_{i,i'} = \begin{cases} t_k & \text{if } \ell_{i,i'} = x_k \\ f_k & \text{if } \ell_{i,i'} = \neg\, x_k \end{cases} \tag{3.1.1}$$

Jobs $j_{i,1}$, $j_{i,2}$, and $j_{i,3}$ can finish before vertex $2n+i+2$ if and only if one of them has completed before vertex $2n+1$. Intuitively, this can occur if and only if the truth assignment chosen by the scheduling algorithm makes one of the literals $\ell_{i,1}$, $\ell_{i,2}$, or $\ell_{i,3}$ true, i.e., if the truth assignment makes ϕ true.

3.1.2 Proof of NP-hardness of *Discrete-time tree scheduling*

We now precisely define the reduction from 3-SAT to *Discrete-time tree scheduling*. We first describe a polynomial-time function that maps each 3-CNF formula ϕ to a tree scheduling problem $P[\phi]$. We then prove that ϕ is satisfiable if and only if $P[\phi]$ has a winning discrete-time strategy. This will establish that *Discrete-time tree scheduling* is NP-hard (Theorem 3.1.3).

Given a 3-CNF formula $\phi = \bigwedge_{i=1}^{m} \ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3}$, we define the conditional scheduling problem $P[\phi] = ((\mathcal{V}, v_0, \mathcal{E}, \mathcal{D}), (J, t, r, d))$ as follows:

- The set $\mathcal{V}$ of vertices is $[1 .. 2n+m+2]$.
- The initial vertex v_0 is 1.
- The set $\mathcal{E}$ of edges is:

$$\begin{array}{lcc} \{(2i-1) \to (2i) \mid i \in [1 .. n+1]\} & \cup & \text{(assignment gadget)} \\ \{(2i-1) \to (2i+1) \mid i \in [1 .. n]\} & \cup & \text{"} \\ \{(2n+1) \to (2n+i+2) \mid i \in [1 .. m]\} & & \text{(formula gadget)} \end{array}$$

- For each edge $e = v \to v' \in \mathcal{E}$, the duration $\mathcal{D}(e)$ of e is defined as follows:

$$\mathcal{D}(e) = \begin{cases} 2 & \text{if } v = 2n+1 \text{ and } v' = 2n+i+2 \text{ for some } i \in [1 .. m] \\ 1 & \text{otherwise} \end{cases}$$

- The set J of jobs is $\{t_1, f_1, t_2, f_2, \cdots, t_n, f_n\}$.
- The time $t(j)$ required by job j is 1 for each job $j \in J$.
- For $i \in [1 .. n]$, the set $r(2i-1)$ of jobs released at vertex $2i-1$ is $\{t_i, f_i\}$. For all other vertices $v \in \mathcal{V}$, $r(v) = \emptyset$.
- The set $d(v)$ of jobs due at each vertex v is defined as follows:

$$\begin{array}{lcll} d(2i+2) & = & \{t_i, f_i\} & \text{for } i \in [1 .. n] \\ d(2n+i+2) & = & \{j_{i,1},\ j_{i,2},\ j_{i,3}\} & \text{for } i \in [1 .. m] \end{array}$$

  where for $i \in [1 .. m]$ and $i' \in [1 .. 3]$, $j_{i,i'}$ is defined by (3.1.1). For all other vertices $v \in \mathcal{V}$, $d(v) = \emptyset$.

It can easily be seen that the conditional scheduling problem $P[\phi]$ can be derived from the 3-CNF formula ϕ in time polynomial in the size of ϕ. We now prove the following lemma, which establishes Theorem 3.1.3.

Lemma 3.1.4. The 3-CNF formula ϕ is satisfiable if and only if the tree scheduling problem $P[\phi]$ has a winning discrete-time strategy.

Proof. ($\Rightarrow$) Suppose that ϕ is satisfiable. Let T be a truth assignment which makes ϕ true. We define the discrete-time strategy $\sigma[T]$ as follows:

- For $i \in [1 \mathrel{..} n]$, if $T(x_i) = \mathit{true}$, then:

$$\begin{aligned}\sigma[T]((1, \cdots, 2i+1), t_i) &= 1\\ \sigma[T]((1, \cdots, 2i+1), f_i) &= 0\\ \sigma[T]((1, \cdots, 2i+2), t_i) &= 0\\ \sigma[T]((1, \cdots, 2i+2), f_i) &= 1\end{aligned}$$

  and if $T(x_i) = \mathit{false}$, then:

$$\begin{aligned}\sigma[T]((1, \cdots, 2i+1), t_i) &= 0\\ \sigma[T]((1, \cdots, 2i+1), f_i) &= 1\\ \sigma[T]((1, \cdots, 2i+2), t_i) &= 1\\ \sigma[T]((1, \cdots, 2i+2), f_i) &= 0\end{aligned}$$

- For $i \in [1 \mathrel{..} m]$:
  - If T makes each literal $\ell_{i,1}$, $\ell_{i,2}$, and $\ell_{i,3}$ true, then $\sigma[T]((1, \cdots, 2n+i+2), j) = 0$ for each $j \in J$.
  - If T makes exactly one literal $\ell_{i,i'}$ false, then $\sigma[T]((1, \cdots, 2n+i+2), j_{i,i'}) = 1$, where $j_{i,i'}$ is defined by (3.1.1). For each other job $j \in J$, $\sigma[T]((1, \cdots, 2n+i+2), j) = 0$.
  - If T makes exactly two literals $\ell_{i,i'}$ and $\ell_{i,i''}$ false, then $\sigma[T]((1, \cdots, 2n+i+2), j_{i,i'}) = \sigma[T]((1, \cdots, 2n+i+2), j_{i,i''}) = 1$, where $j_{i,i'}$ and $j_{i,i''}$ are defined by (3.1.1). For each other job $j \in J$, $\sigma[T]((1, \cdots, 2n+i+2), j) = 0$.

This completes the definition of $\sigma[T]$. We now show that $\sigma[T]$ is winning. By construction, both jobs t_i and f_i finish by vertex $2i+2$. It remains to show that jobs $j_{i,1}$, $j_{i,2}$, and $j_{i,3}$ finish by vertex $2n+2i+2$. Since T makes ϕ true, for each $i \in [1 \mathrel{..} m]$ there is some $i' \in [1 \mathrel{..} 3]$ such that T makes literal $\ell_{i,i'}$ true. The corresponding job $j_{i,i'}$ finishes by vertex $2n+1$. At most two jobs remain to be scheduled before vertex $2n+2i+2$. By the construction of $\sigma[T]$, these other two jobs finish by vertex $2n+2i+2$. Thus, $\sigma[T]$ is winning.

($\Leftarrow$) Suppose that $P[\phi]$ has a winning discrete-time strategy σ. We define the truth assignment $T[\sigma]$ as follows:

$$T[\sigma](x_i) = \begin{cases} \mathit{true} & \text{if } \sigma((1, \cdots, 2i+1), t_i) = 1\\ \mathit{false} & \text{if } \sigma((1, \cdots, 2i+1), f_i) = 1\end{cases}$$

Note that since σ is a winning discrete-time strategy, either $\sigma((1, \cdots, 2i+1), t_i) = 1$ or $\sigma((1, \cdots, 2i+1), f_i) = 1$, but not both. We now show that $T[\sigma]$ makes ϕ true. Let i be an arbitrary member of $[1 \mathrel{..} m]$. Consider the i-th clause of ϕ, $\ell_{i,1} \vee \ell_{i,1} \vee \ell_{i,3}$. Since σ is winning, either $j_{i,1}$, $j_{i,2}$, or $j_{i,3}$ is finished by vertex $2n+1$. Let $i' \in [1 \mathrel{..} 3]$ be such that $j_{i,i'}$ is finished by vertex $2n+1$. Let k be the index of the variable x_k of literal $\ell_{i,i'}$. Without loss of generality suppose that $\ell_{i,i'}$ is positive, so that $j_{i,i'} = t_k$. Since job t_k is finished by vertex $2n+1$, $\sigma((1, \cdots, 2k+1), t_k) = 1$. Thus, $T[\sigma](x_k) = \mathit{true}$, and $T[\sigma]$ makes the i-th clause true. Since i was arbitrary, $T[\sigma]$ makes ϕ true. We conclude that ϕ is satisfiable. □

It should be noted why the construction of $P[\phi]$ fails to show that dense-time tree scheduling is NP-hard. Given the conditional scheduling problem $P[\phi]$, consider a strategy σ such that for $i \in [1 \mathrel{..} n]$:

$$\sigma((1, \cdots, 2i+1), t_i) = \sigma((1, \cdots, 2i+1), f_i) = 0.5$$

In this strategy, every job is executed for 0.5 time units before vertex $2n+1$. The i-th branch of the clause gadget requires that three jobs finish within two time units. A total of 1.5 time units of execution time for these three jobs remains after vertex $2n+1$, so they can easily finish on time.[9]

[9] In fact, this fractional strategy is what initially suggested to the author the polyhedral algorithm for dense-time tree scheduling.

3.2 Directed acyclic graph scheduling

The polynomial-time algorithms from Section 2 all operate on conditional scheduling problems in which the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, is a tree. This raises an obvious question: if the graph $\mathcal{G}$ is allowed to have a more general form — for example, a directed acyclic graph (DAG) — can it still be determined in polynomial time whether a conditional scheduling problem has a winning strategy? In this section, we will show that the answer to this question is negative. This hardness result is unfortunate, as control-flow graphs are, for most programming languages, directed acyclic graphs. Thus, it would be useful to be able to determine whether a DAG scheduling problem has a winning strategy. We now precisely define the DAG scheduling problem.

Definition 3.2.1 (DAG scheduling problem, *DAG scheduling*). A *DAG scheduling problem* is a conditional scheduling problem in which the graph $(\mathcal{V}, \mathcal{E})$ is acyclic. The set *DAG scheduling* is $\{P \mid P$ is a DAG scheduling problem such that there exists a winning strategy for $P\}$. □

In the remainder of this section, we show that *DAG scheduling* is coNP-hard. The basic intuition is that the presence of a DAG leads to a set of runs whose size is exponential in the size of the graph $\mathcal{G}$.

Theorem 3.2.2. *DAG scheduling* is coNP-hard, even if the time $t(j)$ required by each job j is 1.

We prove this theorem by displaying polynomial-time reduction from 3-TAUT to *DAG scheduling*. Recall that a *3-DNF formula*

$$\phi = \bigvee_{i=1}^{m} \ell_{i,1} \wedge \ell_{i,2} \wedge \ell_{i,3}$$

is a disjunction of m *clauses*. Each clause is the conjunction $\ell_{i,1} \wedge \ell_{i,2} \wedge \ell_{i,3}$ of three *literals*. As with 3-CNF formulae, each literal is either a boolean variable x_k or a negated boolean variable $\neg x_k$. We will again let n denote the number of variables $x_1, \cdots, x_n$ appearing in ϕ. The set *3-TAUT* is the set of all 3-DNF formulae that are tautologies. Note that a 3-DNF formula ϕ is a tautology if and only if $\neg\phi$ is unsatisfiable. Note also that

$$\neg \bigvee_{i=1}^{m} \ell_{i,1} \wedge \ell_{i,2} \wedge \ell_{i,3} \quad \text{is equivalent to} \quad \bigwedge_{i=1}^{m} \neg\ell_{i,1} \vee \neg\ell_{i,2} \vee \neg\ell_{i,3}$$

Thus, $\neg\phi$ is equivalent to a 3-CNF formula ψ. Because $\phi \in$ 3-TAUT iff $\neg\phi$ is unsatisfiable iff $\psi \notin$ 3-SAT, 3-TAUT is coNP-complete [Sto76].

3.2.1 Introduction to the reduction from 3-TAUT to *DAG scheduling*

Given 3-DNF formula ϕ, the reduction produces a DAG scheduling problem $P[\phi]$ with the following property: $P[\phi]$ has a winning strategy if and only if ϕ is a tautology. The problem $P[\phi]$ consists of three parts: an *assignment* gadget, a *formula* gadget, and a *tail* gadget. All edges have duration 1 unless otherwise noted. The assignment gadget is pictured in Figure 3.2.1. It consists of $3n+1$ vertices. Consider vertex 1, which has two successors. At the top successor, the jobs $t_{1,1}, \cdots, t_{1,m}$ are released. Each of these jobs has a computation time of 1. At the bottom successor, the jobs $f_{1,1}, \cdots, f_{1,m}$ are released. Intuitively, the top successor corresponds to a truth assignment in which x_1 is *true*, and the bottom successor corresponds to an assignment in which x_1 is *false*. Further, a path from vertex 1 to vertex $3n+1$ corresponds to a truth assignment for the variables $x_1, \cdots, x_n$. Job o has a computation time of 1. Job o is released at vertex 1 and due at vertices 2 and 3. Job o is also released at vertices 2 and 3 and due at vertex 4. In any winning strategy, o is the only job executed from time 0 until time $2n$.

The formula gadget is made up of m *clause* gadgets. Figure 3.2.2 depicts the clause gadget for the i-th clause, $\ell_{i,1} \wedge \ell_{i,2} \wedge \ell_{i,3}$. The first vertex $3n+4i-3$ has three successors, corresponding to the three literals. For $i' \in [1 \,..\, 3]$, job $j_{i,i'}$ is due at the i'-th successor, where

$$j_{i,i'} = \begin{cases} f_{k,i} & \text{if } \ell_{i,i'} = x_k \\ t_{k,i} & \text{if } \ell_{i,i'} = \neg x_k \end{cases} \tag{3.2.1}$$

Note that this definition is the opposite of the definition of $j_{i,i'}$ for the NP-hardness proof (equation (3.1.1)). Vertices $3n+4i-2$, $3n+4i-1$, and $3n+4i$ each release the job o. Recall that job o has computation

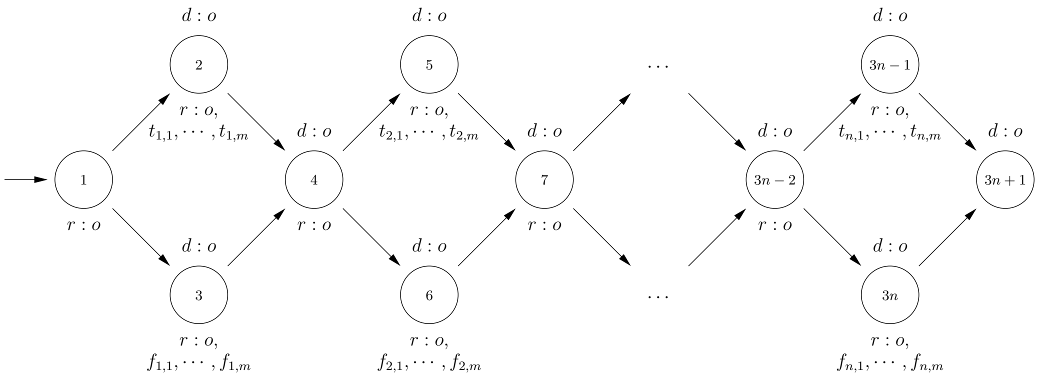

Figure 3.2.1: The assignment gadget.

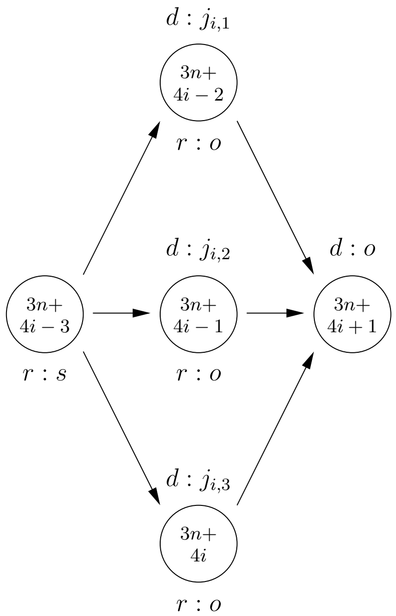

Figure 3.2.2: The clause gadget for the i-th clause.

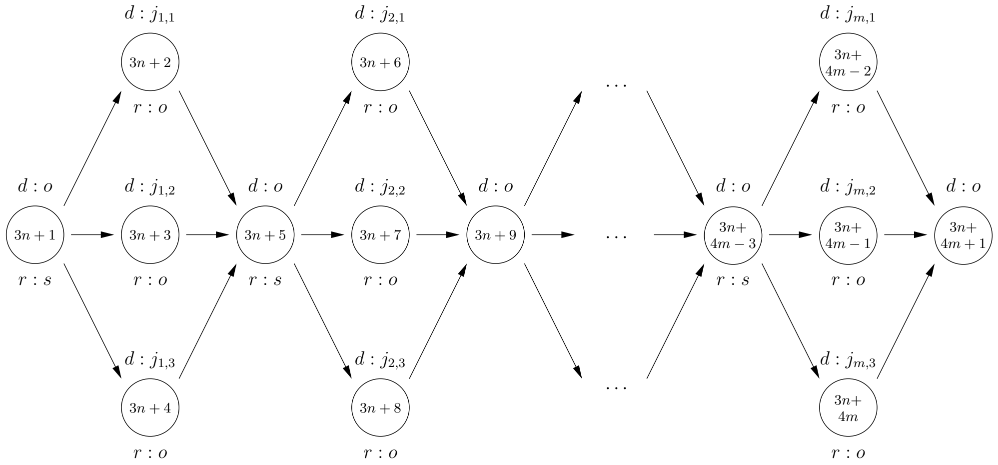

Figure 3.2.3: The formula gadget.

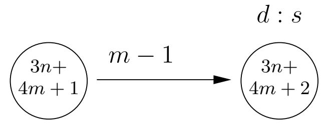

Figure 3.2.4: The tail gadget.

time 1. Job o is due at the common successor vertex $3n + 4i + 1$. In any winning strategy, o is the only job executed from time $2i - 1$ until time $2i$. The vertex $3m + 4i - 3$ releases the job s. Job s has computation time 1, and will be due in the tail gadget. The role of s will be explained shortly. Figure 3.2.3 shows the entire formula gadget. It consists of a sequence of m clause gadgets, one for each clause.

To understand the role of job s, consider a run $\rho = (1, \cdots, 3n+1)$ from vertex 1 to vertex $3n+1$. Let ρ_1 (respectively, ρ_2 and ρ_3) be the sequence consisting of ρ followed by vertex $3n + 2$ (respectively, $3n + 3$, $3n + 4$). There are two cases to consider:

1. For some $i \in [1 .. 3]$, job $j_{1,i}$ has been released along run ρ. Suppose for concreteness that job $j_{1,2}$ has been released. In any winning strategy σ, $\sigma(\rho_2, j_{1,2}) = 1$. Thus, s will not have completed after following ρ_2 and entering vertex $3n + 5$.

2. Neither $j_{1,1}$, $j_{1,2}$, nor $j_{1,3}$ has been released along run ρ. Then s can be executed along runs ρ_1, ρ_2, and ρ_3, and s will have completed upon entering vertex $3n + 5$.

Thus, s will be able complete in all runs following ρ and subsequently entering vertex $3n + 5$ if and only if neither $j_{1,1}$, $j_{1,2}$, nor $j_{1,3}$ has been released along run ρ. But note, from the properties of the assignment gadget, that ρ releases neither $j_{1,1}$, $j_{1,2}$, nor $j_{1,3}$ if and only if the clause $\ell_{1,1} \wedge \ell_{1,2} \wedge \ell_{1,3}$ is made true by the truth assignment corresponding to ρ. Intuitively, then, the first clause gadget indicates whether the first clause is made true by the truth assignment corresponding to ρ: if some amount of s remains to complete upon entering vertex $3n + 5$, then the truth assignment makes the first clause false. Further, if more than $m - 1$ units of s remain to complete upon entering vertex $3n + 4m + 1$, then the truth assignment makes ϕ false. The proof in the next subsection will flesh out these intuitions; here our purpose is merely to introduce the construction.

The tail gadget is pictured in Figure 3.2.4. The tail gadget follows the formula gadget. The tail gadget has two vertices $3n + 4m + 1$ and $3n + 4m + 2$ connected by an edge of duration $m - 1$. The duration of the edge is significant, in that if more than $m-1$ units of job s are pending when vertex $3n+4m+1$ is entered, then s will not be able to finish before vertex $3n+4m+2$ is entered. Intuitively, s will be able to finish before vertex $3n+4m+2$ is entered if and only if every truth assignment makes some clause of ϕ true, i.e., if and only if ϕ is a tautology.

3.2.2 Proof of coNP-hardness of *DAG scheduling*

We now precisely define the reduction from *DAG scheduling* to 3-TAUT. We first present a polynomial-time function that maps each 3-DNF formula ϕ to a DAG scheduling problem $P[\phi]$. We then show that ϕ is a tautology if and only if $P[\phi]$ has a winning strategy, thus establishing that *DAG scheduling* is coNP-hard (Theorem 3.2.2).

Given a 3-DNF formula $\phi = \bigvee_{i=1}^{n} \ell_{i,1} \wedge \ell_{i,2} \wedge \ell_{i,3}$ we define the DAG scheduling problem $P[\phi] = ((\mathcal{V}, v_0, \mathcal{E}, \mathcal{D}), (J, t, r, d))$ as follows:

- The set $\mathcal{V}$ of vertices is $[1 .. 3n+4m+2]$.
- The initial vertex v_0 is 1.
- The set $\mathcal{E}$ of edges is:

$$\begin{array}{lcc}
\{(3i-2) \to (3i-1) \mid i \in [1 .. n]\} & \cup & \text{(assignment gadget)} \\
\{(3i-2) \to (3i) \mid i \in [1 .. n]\} & \cup & \text{''} \\
\{(3i-1) \to (3i+1) \mid i \in [1 .. n]\} & \cup & \text{''} \\
\{(3i) \to (3i+1) \mid i \in [1 .. n]\} & \cup & \text{''} \\
\{(3n+4i-3) \to (3n+4i-2) \mid i \in [1 .. m]\} & \cup & \text{(formula gadget)} \\
\{(3n+4i-3) \to (3n+4i-1) \mid i \in [1 .. m]\} & \cup & \text{''} \\
\{(3n+4i-3) \to (3n+4i) \mid i \in [1 .. m]\} & \cup & \text{''} \\
\{(3n+4i-2) \to (3n+4i+1) \mid i \in [1 .. m]\} & \cup & \text{''} \\
\{(3n+4i-1) \to (3n+4i+1) \mid i \in [1 .. m]\} & \cup & \text{''} \\
\{(3n+4i) \to (3n+4i+1) \mid i \in [1 .. m]\} & \cup & \text{''} \\
\{(3n+4m+1) \to (3n+4m+2)\} & & \text{tail gadget}
\end{array}$$

- For each edge $e = v \to v' \in \mathcal{E}$, the duration $\mathcal{D}(e)$ of e is:

$$\mathcal{D}(e) = \begin{cases} m-1 & \text{if } v = 3n+4m+1 \text{ and } v' = 3n+4m+2 \\ 1 & \text{otherwise} \end{cases}$$

- The set J of jobs is:

$$\begin{array}{ll}
\{t_{i,k} \mid i \in [1 .. n], k \in [1 .. m]\} & \cup \\
\{f_{i,k} \mid i \in [1 .. n], k \in [1 .. m]\} & \cup \\
\{o, s\} &
\end{array}$$

- For each job $j \in J$, the time $t(j)$ required by j is 1.
- The set $r(v)$ of jobs released at vertex $v \in \mathcal{V}$ is defined as follows:

$$\begin{array}{lcll}
r(3i-2) & = & \{o\} & \text{for } i \in [1 .. n] \\
r(3i-1) & = & \{o, t_{i,k} \mid k \in [1 .. m]\} & \text{for } i \in [1 .. n] \\
r(3i) & = & \{o, f_{i,k} \mid k \in [1 .. m]\} & \text{for } i \in [1 .. n] \\
r(3n+4i-3) & = & \{s\} & \text{for } i \in [1 .. m] \\
r(3n+4i-2) & = & \{o\} & \text{for } i \in [1 .. m] \\
r(3n+4i-1) & = & \{o\} & \text{for } i \in [1 .. m] \\
r(3n+4i) & = & \{o\} & \text{for } i \in [1 .. m]
\end{array}$$

  For all other vertices $v \in \mathcal{V}$, $r(v) = \emptyset$.

- The set $d(v)$ of jobs due at vertex $v \in \mathcal{V}$ is defined as follows:

$$\begin{array}{lcll} d(3i-1) & = & \{o\} & \text{for } i \in [1 \,..\, n] \\ d(3i) & = & \{o\} & \text{for } i \in [1 \,..\, n] \\ d(3i+1) & = & \{o\} & \text{for } i \in [1 \,..\, n] \\ d(3n+4i-3+i') & = & \{j_{i,i'}\} & \text{for } i \in [1 \,..\, m]\,, i' \in [1 \,..\, 3] \\ d(3n+4i+1) & = & \{o\} & \text{for } i \in [1 \,..\, m] \\ d(3n+4m+2) & = & \{s\} & \end{array}$$

where $j_{i,i'}$ is defined by (3.2.1). For all other vertices $v \in \mathcal{V}$, $d(v) = \emptyset$.

It can easily be seen that the DAG scheduling problem $P[\phi]$ can be derived from the 3-DNF formula ϕ in time polynomial in the size of ϕ. We now prove the following lemma, which establishes Theorem 3.2.2.

Lemma 3.2.3. The 3-DNF formula ϕ is a tautology if and only if the DAG scheduling problem $P[\phi]$ has a winning strategy.

Proof. ($\Leftarrow$) Suppose that ϕ is not a tautology. We will show that $P[\phi]$ has no winning strategy. Consider a truth assignment T that makes ϕ false. For each clause $\ell_{i,1} \wedge \ell_{i,2} \wedge \ell_{i,3}$ there exists an integer $i' \in [1 \,..\, 3]$ such that T makes $\ell_{i,i'}$ false. Consider the run ρ corresponding to this truth assignment and choice of literals; more precisely, define ρ as follows. For $i \in [1 \,..\, n]$, ρ passes through vertex $3i-1$ if $T(x_i) = \mathit{true}$, or through vertex $3i$ if $T(x_i) = \mathit{false}$. For $i \in [1 \,..\, m]$, ρ passes through vertex $3n+4i-3+i'$. At each of the m vertices $3n+4i-3+i'$, the job $j_{i,i'}$ (as defined by (3.2.1)) is due; moreover each such job has been been released on run ρ. From vertex 1 to $3n+1$ there are $2n$ time units of job o due; from vertex $3n+1$ to $3n+4m+1$ there are m time units of o due and m time units of the jobs $j_{i,i'}$ due, for a total of $2n+2m$ time units. If each instance of job o and each job $j_{i,i'}$ completes in the $2n+2m$ time units before vertex $3n+4m+1$, then the m time units of job s cannot complete in the remaining $m-1$ free time units before s is due at vertex $3n+5m$. Thus $P[\phi]$ does not have a winning strategy.

($\Rightarrow$) Suppose that ϕ is a tautology. We shall exhibit a strategy σ, and then show that it is winning. Let σ be defined as follows:

- Consider any run $\rho = (1, \cdots, i)$ whose final vertex i is in the set $[2 \,..\, 3n+1]$. Then $\sigma(\rho, o) = 1$.
- Let i be a member of $[1 \,..\, m]$, and let i' be a member of $[1 \,..\, 3]$. Let ρ be a run whose final vertex is $3n+4i-3+i'$. If $j_{i,i'}$ has been released along ρ, then $\sigma(\rho, j_{i,i'}) = 1$, where $j_{i,i'}$ is as defined by (3.2.1). Otherwise $\sigma(\rho, s) = 1$.
- For any run $\rho = (1, \cdots, 3n+4i-3)$, where $i \in [1 \,..\, m]$, $\sigma(\rho, o) = 1$.
- Finally, for any run $\rho = (1, \cdots, 3n+4m+2)$ whose final vertex is $3n+4m+2$, $\sigma(\rho, s) = m-1$.

Note that, by construction, all instances of job o, as well as jobs $t_{i,i'}$ and $f_{i,i'}$, complete before they are due. It remains to show that s also completes before it is due. Consider an arbitrary run ρ from vertex 1 to vertex $3n+5m$, and the corresponding truth assignment T. More precisely, $T(x_i) = \mathit{true}$ if ρ passes through vertex $3i-1$, and $T(x_i) = \mathit{false}$ if ρ passes through vertex $3i$. Since ϕ is a tautology, T makes some clause of ϕ, say $\ell_{i,1} \wedge \ell_{i,2} \wedge \ell_{i,3}$, true. For any $i' \in [1 \,..\, 3]$, consider any continuation ρ' of ρ whose final vertex is $3n+4i-3+i'$. Since the job $j_{i,i'}$ due at vertex $3n+4i-3+i'$ has not been released, $\sigma(\rho, s) = 1$. Since s executes for 1 unit of time before vertex $3n+4m+1$, each instance of s completes before it is due at vertex $3n+4m+2$. Thus σ is winning. □

3.3 Discrete-time directed acyclic graph scheduling

If the graph $(\mathcal{V}, \mathcal{E})$ of a conditional scheduling problem P is allowed to be acyclic, and the strategy is required to be discrete-time, then deciding whether P has a winning strategy becomes PSPACE-hard. In this section, we define the set *Discrete-time DAG scheduling* of DAG scheduling problems with a winning discrete-time strategy. We then reduce the PSPACE-hard problem of determining whether a quantified boolean formula has a winning boolean strategy to *Discrete-time DAG scheduling*.

Definition 3.3.1 (*Discrete-time DAG scheduling*). The set *Discrete-time DAG scheduling* is $\{P \mid P$ is a DAG scheduling problem for which there exists a winning discrete-time strategy$\}$. □

In the remainder of this section, we prove the following theorem:

Theorem 3.3.2. *Discrete-time DAG scheduling* is PSPACE-hard, even if the time $t(j)$ required by each job j is 1.

We prove this theorem by presenting a polynomial-time reduction from QBF, the set of all quantified boolean formulae with winning boolean strategies, to *Discrete-time DAG scheduling*. As will be seen, this reduction will combine elements of the reductions of Sections 3.1 and 3.2. For the purposes of this section, a *quantified boolean formula* is a boolean formula of the form $\psi = \exists x_1 \; \forall x_2 \; \cdots \; \exists x_{2n-1} \; \forall x_{2n} \; \phi$, where ϕ is a 3-CNF formula. As before, we shall let m denote the number of clauses of ϕ. We assume that the variables occurring in ϕ are a subset of $\{x_1, \cdots, x_{2n}\}$, and that each clause of ϕ contains three distinct variables. Let A be the set of strings over the alphabet $\{\mathit{true}, \mathit{false}\}$ of length at most $n-1$, i.e., $A = \big\{(\mathit{true} \cup \mathit{false})^i \mid i \in [0 \,..\, n-1]\big\}$. A *boolean strategy* is a function $\beta : A \to \mathbb{B}$. Intuitively, given values $\alpha \in A$ for the universally quantified variables $x_2, x_4, \cdots, x_{2i}$, a boolean strategy chooses a value $\beta(\alpha)$ for the existentially quantified variable x_{2i+1}. Note that $\beta(\lambda)$ is defined, where λ is the empty string, so that a boolean strategy chooses a value for x_1 given values for no universally quantified variables.

Given a boolean strategy β and a string $\alpha = \alpha_1\alpha_2\cdots\alpha_n \in (\mathit{true} \cup \mathit{false})^n$, we define the truth assignment $T[\alpha, \beta]$ as follows. For $i \in [1 \,..\, 2n]$,

$$T[\alpha,\beta](x_i) = \begin{cases} \beta(\alpha_1\alpha_2\cdots\alpha_{\lfloor i/2 \rfloor}) & \text{if } i \text{ is odd} \\ \alpha_{i/2} & \text{if } i \text{ is even} \end{cases}$$

A boolean strategy β is *winning* for the quantified boolean formula ψ if for every string $\alpha \in (\mathit{true} \cup \mathit{false})^n$, the truth assignment $T[\alpha, \beta]$ makes ϕ true. The set *QBF* is $\{\psi \mid \psi$ is a quantified boolean formula for which there exists a winning boolean strategy$\}$. QBF is PSPACE-hard [Sto76].

3.3.1 An introduction to the reduction from QBF to *Discrete-time DAG scheduling*

The expressiveness of the problem *Discrete-time DAG scheduling* allows us to encode the problem of determining whether a quantified boolean formula has a winning boolean strategy as a discrete-time DAG scheduling problem. In this subsection, we introduce the reduction from QBF to *Discrete-time DAG scheduling*. In the next subsection, we precisely define the reduction, and prove that *Discrete-time DAG scheduling* is PSPACE-hard. Given a quantified boolean formula ψ, the reduction from QBF to *Discrete-time DAG scheduling* produces a DAG scheduling problem $P[\psi]$ with the property that ψ has a winning boolean strategy if and only if $P[\psi]$ has a winning discrete-time strategy. The problem $P[\psi]$ has two parts, a *quantifier gadget* and a *formula gadget*. The quantifier gadget consists of an existential quantifier subgadget, followed by a universal quantifier subgadget; this pattern repeats n times. Figure 3.3.1 presents the i-th existential quantifier gadget, for $i \in [1 \,..\, n]$. In an intuitive sense, this gadget represents the symbols $\exists x_{2i-1}$. Vertex $5i-4$ releases two jobs, t_{2i-1} and f_{2i-1}. Each of these jobs requires one unit of computation time, and each job must finish by vertex $5i-2$, two time units later. Since the strategy must be discrete-time, one of these jobs must execute in the first time unit, and, on the edge from vertex $5i-3$ to vertex $5i-2$, the other job must execute in the second time unit. The strategy decides which job to execute first. The reader will note the similarity between Figures 3.3.1 and 3.1.3.

Figure 3.3.2 shows the i-th universal quantifier gadget, for $i \in [1 \,..\, n]$. In an intuitive sense, this gadget represents the symbols $\forall x_{2i}$. Vertex $5i-3$ releases job o. This job requires one unit of computation time, and must finish by vertices $5i-1$ and $5i$, each one time unit later. Vertex $5i-1$ releases the job t_{2i}, and vertex $5i$ releases the job f_{2i}. In contrast to the existential quantifier gadget, the environment — not the scheduler — decides whether to release t_{2i} or rather f_{2i}. Vertices $5i-1$ and $5i$ also release job o, which must finish by vertex $5i+1$, one time unit later. The reader will note the similarity between Figures 3.3.2 and 3.2.1.

Figure 3.3.3 presents the formula gadget. Recall that m is the number of clauses of the 3-CNF formula ϕ. For each $i \in [1 \,..\, m]$, the formula gadget has a vertex $5n+i+1$. At vertex $5n+i+1$, jobs $j_{i,1}$, $j_{i,2}$, and $j_{i,3}$ are

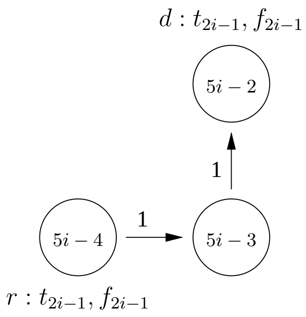

Figure 3.3.1: The existential quantifier subgadget.

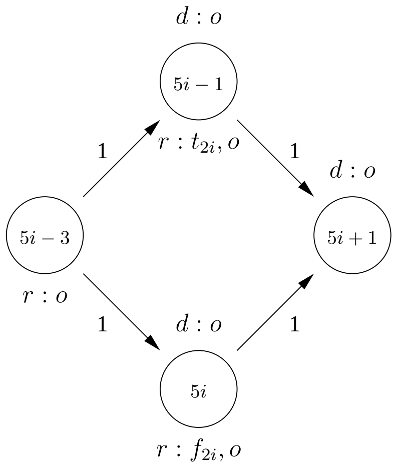

Figure 3.3.2: The universal quantifier subgadget.

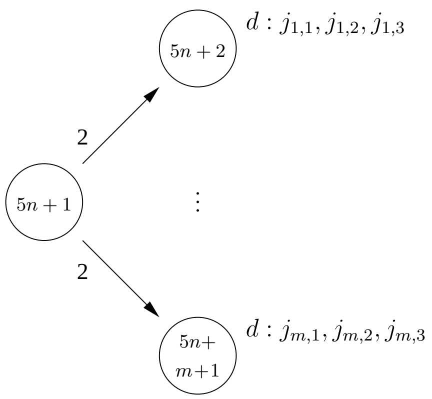

Figure 3.3.3: The formula gadget.

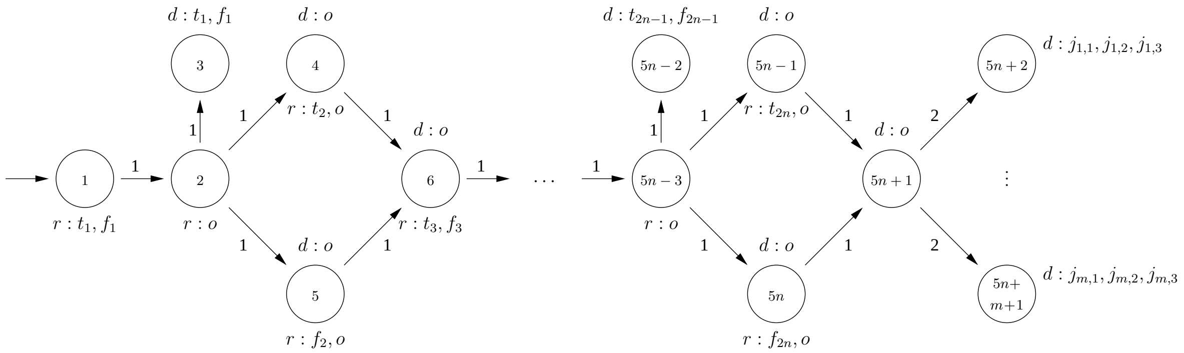

Figure 3.3.4: The entire construction.

due, where for $i' \in [1 .. 3]$, $j_{i,i'}$ is defined by (3.1.1). The reader will note the similarity between Figures 3.3.3 and 3.1.4. Figure 3.3.4 shows the entire construction $P[\psi]$. Intuitively, the choices of the strategy and environment up to time $3n$ (vertex $5n + 1$) correspond to a truth assignment for the variables $x_1, \cdots, x_{2n}$. At time $3n$, the formula gadget then "evaluates" this truth assignment. The proof in the next subsection will flesh out these intuitions.

3.3.2 Proof of PSPACE-hardness

We now precisely describe the reduction from QBF to *Discrete-time DAG scheduling*. First, we present a polynomial-time function that maps each quantified boolean formula ψ to a DAG scheduling problem $P[\psi]$. We then show that ψ has a winning boolean strategy if and only if $P[\psi]$ has a winning discrete-time strategy, thus establishing that *Discrete-time DAG scheduling* is PSPACE-hard (Theorem 3.3.2).

Given a quantified boolean formula $\psi = \exists x_1 \ \forall x_2 \ \cdots \ \exists x_{2n-1} \ \forall x_{2n} \ \bigwedge_{i=1}^{m} \ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3}$, we define the DAG scheduling problem $P[\psi] = ((\mathcal{V}, v_0, \mathcal{E}, \mathcal{D}), (J, t, r, d))$ as follows:

- The set $\mathcal{V}$ of vertices is $[1 .. 5n + m + 1]$.
- The initial vertex v_0 is 1.
- The set $\mathcal{E}$ of edges is:

$$\begin{array}{lcc}
\{(5i-4) \to (5i-3) \mid i \in [1 .. n]\} & \cup & \text{(existential quantifier subgadget)} \\
\{(5i-3) \to (5i-2) \mid i \in [1 .. n]\} & \cup & \text{''} \\
\{(5i-3) \to (5i-1) \mid i \in [1 .. n]\} & \cup & \text{(universal quantifier subgadget)} \\
\{(5i-3) \to (5i) \mid i \in [1 .. n]\} & \cup & \text{''} \\
\{(5i-1) \to (5i+1) \mid i \in [1 .. n]\} & \cup & \text{''} \\
\{(5i) \to (5i+1) \mid i \in [1 .. n]\} & \cup & \text{''} \\
\{(5n+1) \to (5n+i+1) \mid i \in [1 .. m]\} & & \text{(formula gadget)}
\end{array}$$

- For each edge $e = v \to v' \in \mathcal{E}$, the duration $\mathcal{D}(e)$ of e is defined as follows:

$$\mathcal{D}(e) = \begin{cases} 2 & \text{if } v = 5n+1 \text{ and } v' = 5n+i+1 \text{ for some } i \in [1 .. m] \\ 1 & \text{otherwise} \end{cases}$$

- The set J of jobs is $\{o, t_1, f_1, t_2, f_2, \cdots, t_{2n}, f_{2n}\}$.
- The time $t(j)$ required by job j is 1 for each $j \in J$.

- The release function r is defined as follows. For $i \in [1 \,..\, n]$,

$$\begin{array}{lcl} r(5i-4) & = & \{t_{2i-1}, f_{2i-1}\} \\ r(5i-3) & = & \{o\} \\ r(5i-1) & = & \{o, t_{2i}\} \\ r(5i) & = & \{o, f_{2i}\} \end{array}$$

  For each other vertex $v \in \mathcal{V}$, $r(v) = \emptyset$.

- The due function d is defined as follows:

$$\begin{array}{lcll} d(5i-2) & = & \{t_{2i-1}, f_{2i-1}\} & \text{for } i \in [1 \,..\, n] \\ d(5i-1) & = & \{o\} & \text{for } i \in [1 \,..\, n] \\ d(5i) & = & \{o\} & \text{for } i \in [1 \,..\, n] \\ d(5i+1) & = & \{o\} & \text{for } i \in [1 \,..\, n] \\ d(5n+i+1) & = & \{j_{i,1}, j_{i,2}, j_{i,3}\} & \text{for } i \in [1 \,..\, m] \end{array}$$

  where for $i \in [1 \,..\, m]$ and $i' \in [1 \,..\, 3]$, $j_{i,i'}$ is defined by (3.1.1). For each other vertex $v \in \mathcal{V}$, $d(v) = \emptyset$.

It can easily be seen that $P[\psi]$ can be derived from ψ in time polynomial in the size of ψ. We now prove the following lemma, which establishes Theorem 3.3.2.

Lemma 3.3.3. The quantified boolean formula ψ has a winning boolean strategy if and only if the DAG scheduling problem $P[\psi]$ has a winning discrete-time strategy.

Proof. ($\Rightarrow$) Suppose that ψ has a winning boolean strategy β. From β we will construct a winning discrete-time strategy $\sigma[\beta]$. For any run $\rho = (1, \cdots, i) \in R$, where $i \in [2 \,..\, 5n+1]$, define the string $\alpha[\rho] \in (\mathit{true} \cup \mathit{false})^{\lfloor (i+2)/6 \rfloor}$ as follows: $\alpha[\rho] = \alpha[\rho]_1 \cdots \alpha[\rho]_{\lfloor (i+2)/6 \rfloor}$, where for $j \in [1 \,..\, \lfloor (i+2)/6 \rfloor]$,

$$\alpha[\rho]_j = \begin{cases} \mathit{false} & \text{if } \rho \text{ visits vertex } 5j-1 \\ \mathit{true} & \text{if } \rho \text{ visits vertex } 5j \end{cases}$$

Consider any run $\rho = (1, \cdots, k) \in R$. We now define $\sigma[\beta](\rho, j)$ for each job $j \in J$:

- If $k \in [5i-1 \,..\, 5i+1]$ for some $i \in [1 \,..\, n]$, then $\sigma[\beta](\rho, o) = 1$.
- If $k = 5i-3$ for some $i \in [1 \,..\, n]$, then

$$\begin{array}{ll} \sigma[\beta](\rho, t_i) = 1 & \text{if } \beta(\alpha[\rho]) = \mathit{true} \\ \sigma[\beta](\rho, f_i) = 1 & \text{if } \beta(\alpha[\rho]) = \mathit{false} \end{array}$$

  If $k = 5i-2$ for some $i \in [1 \,..\, n]$, then

$$\begin{array}{ll} \sigma[\beta](\rho, t_i) = 1 & \text{if } \beta(\alpha[\rho]) = \mathit{false} \\ \sigma[\beta](\rho, f_i) = 1 & \text{if } \beta(\alpha[\rho]) = \mathit{true} \end{array}$$

- If $k = 5n+i+1$ for some $i \in [1 \,..\, m]$, let ρ' be the prefix of ρ that agrees with ρ up to vertex $5n+1$. Since $\psi \in$ QBF, the truth assignment $T[\alpha[\rho'], \beta]$ makes false at most two of the literals $\ell_{i,1}$, $\ell_{i,2}$, and $\ell_{i,3}$.
  - If $T[\alpha[\rho'], \beta]$ makes none of these literals false, then $\sigma[\beta](\rho, j) = 0$ for each $j \in J$.
  - If $T[\alpha[\rho'], \beta]$ makes exactly one literal $\ell_{i,i'}$ false, then $\sigma[\beta](\rho, j_{i,i'}) = 1$, where $j_{i,i'}$ is defined by (3.1.1). For each other job $j \in J$, $\sigma[\beta](\rho, j) = 0$.
  - If $T[\alpha[\rho'], \beta]$ makes exactly two literals $\ell_{i,i'}$ and $\ell_{i,i''}$ false, then $\sigma[\beta](\rho, j_{i,i'}) = \sigma[\beta](\rho, j_{i,i''}) = 1$, where $j_{i,i'}$ and $j_{i,i''}$ are defined by (3.1.1). For each other job $j \in J$, $\sigma[\beta](\rho, j) = 0$.

We now claim that $\sigma[\beta]$ is winning. By the construction of $\sigma[\beta]$, all deadlines are met at vertices $[5i-2 \mathrel{..} 5i+1]$, for all $i \in [1 \mathrel{..} n]$. Now consider any $i \in [1 \mathrel{..} m]$, and any run ρ from vertex 1 to vertex $5n+i+1$. It is easy to see, based on the fact that β is a winning boolean strategy, that at most two of the jobs $j_{i,1}$, $j_{i,2}$, and $j_{i,3}$ remain to complete upon entering vertex $5n+1$. By the construction of $\sigma[\beta]$, these jobs are all complete upon entering vertex $5n+i+1$. Thus $\sigma[\beta]$ is winning.

($\Leftarrow$) Suppose that $P[\psi]$ has a winning discrete-time strategy σ. From σ we will construct a winning boolean strategy $\beta[\sigma]$. Given an integer $i \in [0 \mathrel{..} n-1]$ and a string $\alpha = \alpha_1 \cdots \alpha_i \in (\mathit{true} \cup \mathit{false})^i$, let $\rho[\alpha] = \rho[\alpha]_1 \cdots \rho[\alpha]_{3i+2}$ be the unique run from vertex 1 to vertex $5i+2$ such that for $j \in [1 \mathrel{..} i]$,

$$\rho[\alpha]_{3j} = \begin{cases} 5j-1 & \text{if } \alpha_j = \mathit{false} \\ 5j & \text{if } \alpha_j = \mathit{true} \end{cases}$$

Define $\beta[\sigma]$ as follows:

$$\beta[\sigma](\alpha) = \begin{cases} \mathit{true} & \text{if } \sigma(\rho[\alpha], t_{2i+1}) = 1 \\ \mathit{false} & \text{if } \sigma(\rho[\alpha], f_{2i+1}) = 1 \end{cases}$$

We now show that the strategy $\beta[\sigma]$ is winning. Consider any string $\alpha \in (\mathit{true} \cup \mathit{false})^n$ and any integer $i \in [1 \mathrel{..} m]$. We will show that the truth assignment $T[\alpha, \beta[\sigma]]$ makes true the i-th clause, $\ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3}$. Since σ is winning, after following the run $\rho[\alpha_1 \cdots \alpha_{n-1}]$ and entering vertex $5n+1$, at most two of the jobs $j_{i,1}$, $j_{i,2}$, and $j_{i,3}$ remain to complete. Based on this fact, it is easy to see that at most two of the literals $\ell_{i,1}$, $\ell_{i,2}$, and $\ell_{i,3}$ are made false by the truth assignment $T[\alpha, \beta[\sigma]]$. Thus $T[\alpha, \beta[\sigma]]$ makes the i-th clause true, and $\beta[\sigma]$ is winning. □

4 Fixed-deadline conditional scheduling

For conditional scheduling problems in which the graph $(\mathcal{V}, \mathcal{E})$ contains cycles, developing a strategy synthesis algorithm — even an exponential-time algorithm — has proved very difficult. Though the author cannot locate the source of this difficulty with complete confidence, he nonetheless believes that the problem lies with the lack of *standard forms* of winning strategies for conditional scheduling problems. This concept is best explained by comparison with the EDF scheduling algorithm. The way that EDF is shown to optimal, say for the problem $1 \mid r_j;\ d_j;\ \mathit{prec};\ \mathit{pmtn} \mid -$, is to show that an arbitrary feasible schedule S can be rearranged into the schedule S' that EDF would have produced. The schedule produced by EDF is thus a standard form that represents a class of feasible schedules. For a conditional scheduling problem P, the process of going from the constraints $\mathit{Lin}[P]$ to a winning strategy σ involve global optimization, so that one is hard-pressed to locate a standard form for winning strategies.

Suppose however that P is a conditional scheduling problem in which all deadlines are fixed. That is, after a job j is released, j has a deadline some fixed number of time units later. This deadline is the same regardless of which vertex releases j, and regardless of the sequence of vertices encountered after j is released. It was observed in [CET01] that, for such a *fixed-deadline* conditional scheduling problem, EDF is an optimal scheduling algorithm. The technical challenge now becomes one of detecting whether a condition of overload obtains. In this section, we will develop an algorithm for this purpose. The algorithm that we obtain will run in pseudopolynomial-time, and will determine whether a fixed-deadline conditional scheduling problem has a winning strategy. This work extends that of [CET01] in that the type of graph $(\mathcal{V}, \mathcal{E})$ that we analyze is much more general: in [CET01] the graph is required to be a DAG with a unique source vertex and a unique sink vertex. These restrictions may be slightly relaxed by allowing an edge from the sink to the source, the only cycles in the graph being formed by such an edge [Cha03]. In this section, we remove these restrictions. The cost is an increase in complexity: whereas [CET01] presented a fully polynomial-time approximation scheme, our algorithm is a pseudopolynomial-time exact algorithm. There is some hope that this shortcoming might be removable, as the approximation scheme of [CET01] was itself derived from a pseudopolynomial-time exact algorithm.

We begin with a definition of fixed-deadline conditional scheduling. Rather than placing additional conditions on Definition 2.0.1 to obtain a definition of fixed-deadline problems, we opt for the following, less cumbersome, definition.

Definition 4.0.4 (fixed-deadline conditional scheduling problem). A *fixed-deadline conditional scheduling problem* is a pair $(\mathcal{F}, W)$, where:

- $\mathcal{F}$ is defined as in Definition 2.0.1.
- $W = (J, t, r, d)$, where J, t, and r are defined as in Definition 2.0.1, and $d : J \to \mathbb{Q}^{>0}$ is a function assigning to each job j a positive rational *deadline* $d(j)$. □

We now define the class of fixed-deadline conditional scheduling problems that our algorithm is capable of analyzing.

- We require that the duration $\mathcal{D}(e)$ of each edge e is an integer, and that the time $t(j)$ and deadline $d(j)$ of each job j are integers. This requirement may be removed by multiplying each of these quantities by the least common multiple of the denominators of these quantities; for simplicity, however, we retain this requirement.
- We require that every vertex have a successor. This requirement is not strict, as any graph not satisfying it may be transformed into a graph that does: add an edge from every vertex with no successor to a new vertex v that releases no jobs, with a new edge from v back to itself.
- We require that if a vertex releases a job j, then at least $d(j)$ time units pass before j is released again. Unlike the previous two requirements, this one is necessary for the algorithm we shall develop.

More precisely, we define as follows the conditions under which a fixed-deadline conditional scheduling problem is well-formed.

Definition 4.0.5 (well-formed fixed-deadline conditional scheduling problem). A fixed-deadline conditional scheduling problem is *well-formed* if the following conditions hold:

- For every edge $e \in \mathcal{E}$, $\mathcal{D}(e) \in \mathbb{Z}^{>0}$, and for every job $j \in J$, $t(j), d(j) \in \mathbb{Z}^{>0}$.
- For each $v \in \mathcal{V}$ there exists a vertex $v' \in \mathcal{V}$ such that $v \to v'$.
- Consider any job $j \in J$ and any run $\rho = (v_0, \cdots, v_n) \in R$. Let $i < k$ be two nonnegative integers such that $j \in r(v_i)$ and $j \in r(v_k)$. Then $d(j) \leq \sum_{\ell=i}^{k-1} \mathcal{D}(v_i \to v_{i+1})$. □

Because numerical quantities rather than vertices are now used to impose deadlines, exactly when a job executes between $\tau(i)$ and $\tau(i+1)$ is now important, rather than just the proportion of time allocated to a job between $\tau(i)$ and $\tau(i+1)$. For this reason, the definition of a strategy has to be modified to fit our new setting.

Definition 4.0.6 (strategy). Let P be a well-formed fixed-deadline conditional scheduling problem. A strategy σ for P is a function that assigns to each run $\rho = (v_0, \cdots, v_n) \in R$ a pair $\sigma(\rho) = (I, e)$ such that:

- I is a set of intervals, each of which is a nonempty, left-open and right-open subset of $(\tau(v_{n-1}), \tau(v_n))$. Distinct intervals must not overlap, i.e., if $i, i' \in I$ and $i \neq i'$, then $i \cap i' = \emptyset$.
- $e : I \to J$ is a function mapping each interval i to a job $e(i)$. We say that job $e(i)$ is *executed* in interval i.

For a strategy σ and a run $\rho \in R$, we let $I[\sigma, \rho]$ (respectively, $e[\sigma, \rho]$) denote I (respectively, e), where $(I, e) = \sigma(\rho)$. Intuitively, $I[\sigma, \rho]$ are the intervals in which some job is executed by σ along ρ, and $e[\sigma, \rho]$ is a function which gives the jobs executed in these intervals. Finally, for a job $j \in J$, we let $I[\sigma, \rho, j]$ denote the set $\{i \in I[\sigma, \rho] \mid j = e[\sigma, \rho](i)\}$. Intuitively, $I[\sigma, \rho, j]$ is the set of intervals in which job j is executed by σ along ρ. □

We now define the conditions under which a strategy is winning. A strategy σ is winning if, for every infinite path $(v_0, v_1, \cdots)$ through the graph $(\mathcal{V}, \mathcal{E})$, for every integer $i \in \mathbb{Z}^{\geq 0}$, for every job $j \in r(v_i)$, σ allocates $t(j)$ time to job j between $\tau(i)$ and $\tau(i) + d(j)$. More precisely:

Algorithm 4.0.1 EDF algorithm for fixed-deadline conditional scheduling problems.

1: Algorithm EDF(Time: $\mathbb{Z}^{\geq 0}$, $\mathit{Remaining}$: array J of $\mathbb{Z}^{\geq 0}$, $\mathit{Deadline}$: array J of $\mathbb{Z}^{\geq 0}$)
2: **while** $\mathit{Time} > 0$ and $\mathit{Remaining}[j] > 0$ for some $j \in J$ **do**
3:     **let** $j :=$ a job in J such that $\mathit{Deadline}[j]$ is minimal **in**
4:         $\Delta := \min\{\mathit{Time}, \mathit{Remaining}[j]\}$
5:         $\mathit{Time} := \mathit{Time} - \Delta$, $\mathit{Remaining}[j] := \mathit{Remaining}[j] - \Delta$
6:         **for all** $j \in J$ such that $\mathit{Remaining}[j] > 0$ **do**
7:             $\mathit{Deadline}[j] := \mathit{Deadline}[j] - \Delta$
8:             **if** $\mathit{Remaining}[j] > \mathit{Deadline}[j]$ **then**
9:                 P has no winning strategy.

Definition 4.0.7 (winning strategy). A strategy σ for a well-formed fixed-deadline conditional scheduling problem P is *winning* if the following condition holds. Let $(v_0, v_1, \cdots)$ be any infinite sequence of vertices, beginning with the initial vertex v_0, such that $v_i \to v_{i+1}$ for $i \in \mathbb{Z}^{\geq 0}$. Then for any integer $k \in \mathbb{Z}^{\geq 0}$, for any job $j \in r(v_k)$,

$$\sum_{\ell=k+1}^{\infty} \sum_{i \in I[\sigma,(v_0,\cdots,v_\ell),j]} |i \cap (-\infty, \tau(v_k) + d(j))| \quad \geq \quad t(j) \qquad \square$$

From the optimality of EDF for fixed-deadline conditional scheduling [CET01], it may easily be verified that if a well-formed fixed-deadline conditional scheduling problem P has a winning strategy, then P has a winning strategy σ in which the endpoints of each interval $i \in I[\sigma, \rho]$ are integers, for each run $\rho \in R$.

We now develop an algorithm for determining whether a fixed-deadline conditional scheduling problem has a winning strategy. We first present an EDF scheduling algorithm (Algorithm 4.0.1). We then we present an algorithm that uses Algorithm 4.0.1 to detect overload (Algorithm 4.0.2). For any run $(v_0, v_1, \cdots)$, and any $i \in [0 \,..\, \infty]$, Algorithm 4.0.1 describes how EDF schedules the time interval $(\tau(i), \tau(i+1))$ from the time vertex v_i is entered until the time vertex v_{i+1} is entered. EDF has three inputs:

1. *Time*, a nonnegative integer, initially set to the duration of the edge $v_i \to v_{i+1}$. *Time* indicates the amount of time remaining in the interval $(\tau(i), \tau(i+1))$.

2. *Remaining*, an array of nonnegative integers indexed by the set J of jobs, that specifies how many units of execution time remain for each job.

3. *Deadline*, also an array of nonnegative integers indexed by J, that specifies for each job j the amount of time until j's deadline expires.

On lines 2–3, while some time from the interval $(\tau(i), \tau(i+1))$ is remaining, and some job has execution time remaining, a job j with minimal deadline is selected. This job is executed for *Time* or $\mathit{Remaining}[j]$ time units, whichever quantity is smaller; call this quantity Δ (lines 4–5). Line 7 fixes up the array *Deadline* to account for the fact that Δ units of time have elapsed. If there is some job j such that $\mathit{Remaining}[j] > \mathit{Deadline}[j]$, the deadline for j cannot be met (line 8). In this case, P has no winning strategy (line 9).

Unfortunately, since the number of runs, and the lengths of each run, are in general infinite, Algorithm 4.0.1 cannot be used to check each run in turn, as the process would never terminate. However, there are only finitely many permissible vectors *Remaining* and *Deadline*, since

$$\mathit{Remaining}[j] \in [0 \,..\, t(j)] \text{ and } \mathit{Deadline}[j] \in [0 \,..\, d(j)] \text{ for each job } j \in J \tag{4.0.1}$$

This observation suggests a state space exploration algorithm (Algorithm 4.0.2), where a *state* is a pair $(\mathit{Remaining}, \mathit{Deadline})$ of vectors satisfying (4.0.1). For each vertex v, Algorithm 4.0.2 keeps track of a set of unexplored states $\mathit{Frontier}[v]$ and a set of explored states $\mathit{Explored}[v]$. Initially, $\mathit{Frontier}[v_0]$ is just a pair of zero vectors, corresponding to the fact that no jobs have either remaining execution time or upcoming deadlines, and $\mathit{Frontier}[v] = \emptyset$ for each vertex $v \neq v_0$ (lines 2–5). Now, as long as some vertex v has a nonempty frontier, a pair $(\mathit{Remaining}, \mathit{Deadline})$ is selected from $\mathit{Frontier}[v]$ (lines 6–8). This pair is removed from the frontier of v, and added to the explored set of v (lines 9–10). Lines 11–12 update

Algorithm 4.0.2 State-space exploration algorithm for fixed-deadline conditional scheduling.

1: Algorithm Explore(P: well-formed fixed-deadline conditional scheduling problem)
2: **for all** $v \in \mathcal{V}$ **do**
3:     $\mathit{Frontier}[v] := \mathit{Explored}[v] := \emptyset$
4: **let** $\mathit{Remaining}[j] := \mathit{Deadline}[j] := 0$ for each $j \in J$ **in**
5:     $\mathit{Frontier}[v_0] := (\mathit{Remaining}, \mathit{Deadline})$
6: **while** $\mathit{Frontier}[v] \neq \emptyset$ for some $v \in \mathcal{V}$ **do**
7:     Select any vertex v such that $\mathit{Frontier}[v] \neq \emptyset$.
8:     **let** $(\mathit{Remaining}, \mathit{Deadline})$ be any member of $\mathit{Frontier}[v]$ **in**
9:         $\mathit{Frontier}[v] := \mathit{Frontier}[v] \setminus \{(\mathit{Remaining}, \mathit{Deadline})\}$
10:         $\mathit{Explored}[v] := \mathit{Explored}[v] \cup \{(\mathit{Remaining}, \mathit{Deadline})\}$
11:         **for all** $j \in r(v)$ **do**
12:             $\mathit{Remaining}[j] := t(j)$, $\mathit{Deadline}[j] := d(j)$
13:         **for all** $v' \in \mathcal{V}$ such that $v \to v'$ **do**
14:             **let** $(\Delta', \mathit{Remaining}', \mathit{Deadline}') := \mathrm{EDF}(\mathcal{D}(v \to v'), \mathit{Remaining}, \mathit{Deadline})$ **in**
15:                 **if** $(\mathit{Remaining}', \mathit{Deadline}') \notin \mathit{Explored}(v')$ **then**
16:                     $\mathit{Frontier}(v') := \mathit{Frontier}(v') \cup \big\{(\mathit{Remaining}', \mathit{Deadline}')\big\}$
17: **if** EDF has not declared that P has no winning strategy **then**
18:     P has a winning strategy.

Remaining and *Deadline* to account for the jobs released at vertex v. Now, for each successor vertex v' of v, we let $\mathit{Remaining}'$ and $\mathit{Deadline}'$ be the result of applying Algorithm EDF for $\mathcal{D}(v \to v')$ time units (lines 13–14).[10] If $(\mathit{Remaining}', \mathit{Deadline}')$ is not in the explored set of v', then $(\mathit{Remaining}', \mathit{Deadline}')$ is added to the frontier set. Finally, after all reachable states have been visited (line 17), if no call to Algorithm EDF reported failure, then the fixed-deadline problem P is reported to have a winning strategy (line 18).

It is easy to show, based on the optimality of EDF for fixed-deadline conditional scheduling [CET01], that Algorithm 4.0.2 correctly determines whether P has a winning strategy. Further, the while loop in lines 6–16 executes at most once for each vertex v and state (*Deadline*, *Remaining*), i.e., at most $|\mathcal{V}| \left(\prod_{j \in J} t(j)\right) \left(\prod_{j \in J} d(j)\right)$ times. The running time of Algorithm 4.0.2 is thus exponential, but pseudopolynomial, in the size of P. We summarize this result in the following theorem.

Theorem 4.0.8. Algorithm 4.0.2 determines whether a well-formed fixed-deadline conditional scheduling problem P has a winning strategy. The running time of Algorithm 4.0.2 is exponential, and pseudopolynomial, in the size of P.

5 Conclusion

This report has studied conditional scheduling problems. Table 5.0.5 summarizes our results. For conditional scheduling problems with varying deadlines, we located the dividing line between feasible and infeasible versions of conditional scheduling. On the positive side, when the graph $(\mathcal{V}, \mathcal{E})$ is a tree, we developed a polynomial-time algorithm for synthesizing a winning strategy if one exists (Section 2). We extended this algorithm to maximize an anytime reward function (Section 2.2), and to handle precedence constraints (Section 2.3). On the negative side, we showed that under the reasonable restriction that preemptions occur only at integral times, determining whether a tree scheduling problem has a winning strategy is NP-hard (Section 3.1). Also on the negative side, we showed that if $(\mathcal{V}, \mathcal{E})$ is a directed, acyclic graph (DAG), then determining whether a conditional scheduling problem has a winning strategy is coNP-hard (Section 3.2). Further, when DAG scheduling is combined with preemptions only at integral times, determining whether a conditional scheduling problem has a winning strategy becomes PSPACE-hard (Section 3.3). Finally, for conditional scheduling problems with fixed deadlines, we presented a pseudopolynomial-time algorithm that tests for the existence of a winning strategy (Section 4).

[10] We assume a call-by-value semantics, so that *Remaining* and *Deadline* are not modified by the call in line 14 to Algorithm EDF.

Problem	Form of graph	Form of trace tree	Additional features	Result
Tree scheduling	Tree	Dense-time		Polynomial-time algorithm
Imprecise tree scheduling	Tree	Dense-time	Anytime reward function	Polynomial-time algorithm
Precedence-constrained tree scheduling	Tree	Dense-time	Precedence relation	Polynomial-time algorithm
Discrete-time tree scheduling	Tree	Discrete-time		NP-hard
DAG scheduling	Directed, acyclic	Dense-time		coNP-hard
Discrete-time DAG scheduling	Directed, acyclic	Discrete-time		PSPACE-hard
Fixed-deadline conditional scheduling	Any	Discrete- or dense-time	Fixed deadlines	Exponential-time algorithm

Figure 5.0.5: Main results of this report.

References

[AMMMA01] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez. Optimal reward-based scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 50(2):111–130, 2001.

[Bar98a] S.K. Baruah. Feasibility analysis of recurring branching tasks. In *Proc. 10th EUROMICRO Workshop on Real-Time Systems*, pages 138–145. IEEE, 1998.

[Bar98b] S.K. Baruah. A general model for recurring real-time tasks. In *Proc. 19th IEEE Real-Time Systems Symposium*, pages 114–122. IEEE, 1998.

[Bła76] J. Błażewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In *Proc. Intl. Workshop on Modelling and Performance Evaluation of Computer Systems*, pages 57–65. North-Holland, 1976.

[CET01] S. Chakraborty, T. Erlebach, and L. Thiele. On the complexity of scheduling conditional real-time code. In *Proc. Workshop on Algorithms and Data Structures*, pages 38–49. Springer-Verlag, 2001.

[Cha03] S. Chakraborty. Personal communication, 2003.

[DB88] T.L. Dean and M. Boddy. An analysis of time-dependent planning. In *Proc. AAAI 1988*, pages 49–54. Morgan Kaufmann, 1988.

[EKP+98] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. In *Proc. Design, Automation and Test in Europe*, pages 132–138. IEEE, 1998.

[Foh94] G. Fohler. *Flexibility in Statically Scheduled Hard Real Time Systems*. PhD thesis, Vienna University of Technology, April 1994.

[GS98] F. Gasperoni and U. Schwiegelshohn. List scheduling in the presence of branches. a theoretical evaluation. *Theoretical Computer Science*, 196(1–2):347–363, 1998.

[LLSY91] J.W.S. Liu, K. Lin, W. Shih, and A.C. Yu. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, 1991.

[Pap94] C. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[RW91] S.J. Russell and E.H. Wefald. *Do the Right Thing: Studies in Limited Rationality*. MIT Press, 1991.

[SLC91] W. Shih, J.W.S. Liu, and J. Chung. Algorithms for scheduling imprecise computations with timing constraints. *SIAM Journal on Computing*, 20(3):537–552, 1991.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[SRLR89] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):243–264, 1989.

[Sto76] L.J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976.